THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par *l'Université Toulouse II - Le Mirail et L'Université de Sfax*
dans le cadre d'une cotutelle internationale

Discipline ou spécialité : *Informatique*

Présentée et soutenue par *Fatma Krichen*
Le *16/09/2013*

Titre : *Architectures logicielles à composants reconfigurables pour les systèmes Temps Réel Répartis Embarqués (TR²E)*

JURY

Mohamed Abid	*Professeur, Université de Sfax*	*Examinateur*
Bernard Coulette	*Professeur, Université de Toulouse*	*Directeur*
Brahim Hamid	*Maître de conférences, Université de Toulouse*	*Co-directeur*
Mohamed Jmaiel	*Professeur, Université de Sfax*	*Directeur*
Mohamed Mosbah	*Professeur, Université Bordeaux 1*	*Rapporteur*
Frank Singhoff	*Professeur, Université de Bretagne Occidentale*	*Rapporteur*
Bechir Zalila	*Maître assistant, Université de Sfax*	*Co-directeur*

Ecole doctorale : *Mathématiques Informatique Télécommunications (MITT)*
Unité de recherche : *Institut de Recherche en Informatique de Toulouse (IRIT)*
Directeur(s) de Thèse : *Bernard Coulette & Brahim Hamid – Université de Toulouse*
Mohamed Jmaiel & Bechir Zalila – Université de Sfax
Rapporteurs : *Mohamed Mosbah & Frank Singhoff*

À ma mère Naima et mon père Fathi,
À mon mari Slim,
À mon ange, ma petite fille Yousr,
À ma sœur Hajer,
À mes frères Sarhan, Riadh et Mohamed,
À ma belle-mère Nedra et mon beau-père Mohamed,
Aux deux familles Krichen et Kallel,
À tous ceux qui me sont chers.

REMERCIEMENTS

C'est avec un grand plaisir que je réserve cette page en signe de gratitude et de profonde reconnaissance à tous ceux qui ont bien voulu apporter l'assistance nécessaire au bon déroulement de ce travail.

Je tiens, tout d'abord, à remercier mes deux directeurs de thèse Bernard Coulette, professeur à l'Université de Toulouse 2 Le Mirail, et Mohamed Jmaiel, professeur à l'École Nationale d'Ingénieurs de Sfax, qui n'ont pas épargné leurs efforts dans la direction de cette thèse et qui ont été toujours disponibles pour écouter, discuter et m'orienter pour prendre les bonnes décisions.

Je tiens à remercier vivement Brahim Hamid, maitre de conférences à l'Université de Toulouse 2 Le Mirail, et Bechir Zalila, maitre assistant à l'École Nationale d'Ingénieurs de Sfax. Je ne le fais pas parce qu'un doctorant doit (normalement) remercier ces encadrants. Je le fais parce qu'il ne pourrait en être autrement. Je les remercie pour les nombreux conseils qu'ils m'ont donnés, les remarques qu'ils m'ont faites et parce qu'ils ont participé énormément à la réalisation de ce présent travail.

Ensuite, je remercie Frank SINGHOFF et Mohamed MOSBAH, d'avoir accepté d'être les rapporteurs de ce travail. Les remarques pertinentes qu'ils ont émises ont permis de le consolider. J'exprime également ma profonde gratitude à Mohamed ABID qui a accepté d'être membre de mon jury de thèse.

Effectuer cette thèse au sein des deux équipes de recherche MACAO (IRIT) et ReDCAD (ENIS) m'a donné l'occasion de faire la connaissance de plusieurs amis. Le temps passé avec ces amis ainsi que les nombreuses discussions, souvent vives et intéressantes, ont constitué la cerise sur le gâteau de ces années de thèse.

Enfin, je tiens à exprimer ma reconnaissance et mon affection à ma famille pour leur soutien et leur présence tout au long de cette thèse. La totalité de ma reconnaissance et de mes pensées vont à mes parents Fathi et Naima et à mon mari Slim. Ils n'ont jamais cessé de m'encourager et de m'apporter du support. C'est donc tout naturellement que ce document leur est dédié.

Résumé

Le développement des systèmes Temps Réel Répartis Embarqués TR^{2}E dynamiquement reconfigurables est une tâche complexe exigeant d'importants efforts en vue de son bon accomplissement. A travers ce travail de thèse, nous proposons une approche dirigée par les modèles afin de concevoir des systèmes TR^{2}E dynamiquement reconfigurables. Il s'agit de mettre en place une approche permettant, à travers un modèle de haut niveau d'abstraction et des transformations successives, d'obtenir un modèle raffiné proche du code, qui permet ensuite la génération d'une grande partie du code du système.

Notre approche est présentée sous la forme d'un processus de développement permettant de spécifier les reconfigurations dynamiques ainsi que la partie logicielle du système à un haut niveau d'abstraction. Ce processus permet aussi de définir la partie matérielle et l'allocation de la partie logicielle sur celle-ci. En vue d'assurer la construction correcte des systèmes TR^{2}E dynamiquement reconfigurables, notre démarche aide à la vérification de certaines propriétés non-fonctionnelles au niveau modèle.

A l'inverse de MARTE (Modeling and Analysis of Real Time and Embedded systems) et AADL (Architecture Analysis and Design Language), nous proposons une approche permettant la spécification des systèmes TR^{2}E dynamiquement reconfigurables sans énumérer toutes les configurations possibles du système. Pour cela, le concept de MetaMode a été introduit pour capturer et caractériser un ensemble de configurations plutôt que de définir chacune d'entre elles. Un MetaMode est défini par un ensemble de composants structurés, des connexions ainsi que des contraintes structurelles et non-fonctionnelles. Les reconfigurations dynamiques sont spécifiées et décrites par des machines à états composées de MetaModes et de transitions entre eux.

Nous avons conçu un ensemble de méta-modèles, des règles de transformations, des algorithmes de vérification, une plate-forme d'exécution et une suite d'outils pour supporter ces artifacts tout au long du processus de développement. Les méta-

modèles permettent de spécifier la reconfiguration, le système et la plate-forme d'exécution. Les règles de transformation permettent de raffiner les modèles pour aider à la construction du système. Les algorithmes de vérification garantissent la satisfaisabilité des propriétés non-fonctionnelles. La plate-forme d'exécution offre des routines pour supporter les activités de reconfiguration dynamique et la supervision, et assurer la cohérence

Pour aider l'automatisation du développement des systèmes TR^{2}E dynamiquement reconfigurables, nous avons introduit une architecture d'implantation et une suite d'outils pour accompagner le processus proposé. La suite d'outils est composée d'un ensemble d'éditeurs de modèles, d'un framework de vérification, de générateurs de code et d'un intergiciel. L'éditeur de modèle est une extension de l'éditeur UML Papyrus et de ses profils. Les générateurs implantent des transformations de modèles en utilisant les technologies ATL (M2M) et Acceleo (M2T). La vérification est proposée sous la forme d'un plug-in ECLIPSE intégrant le framework Cheddar, alors que l'intergiciel est une extension de PolyORB_HI.

Mots-clé : Reconfiguration dynamique, Systèmes embarqués répartis, Vérification, Plate-forme d'éxecution, Ingénierie dirigée par les modèles, Intergiciel, UML, MARTE.

Table des matières

Table des figures

Table des listings

Liste des tableaux

Liste des tableaux

Chapitre 1

Introduction générale

1.1 Contexte général

Les systèmes embarqués sont devenus indispensables dans notre vie quotidienne. Ils connaissent un essor considérable et envahissent les différents domaines d'application tels que l'avionique, la télécommunication, le transport, etc. Un système embarqué présente une intégration entre deux parties, logicielle et matérielle, qui sont conçues conjointement pour répondre à des fonctionnalités spécifiques. Ces fonctionnalités sont, pour la plupart des cas, des applications critiques [21]. Les systèmes temps réel sont des systèmes ayant des contraintes temporelles strictes et dans lesquels l'exactitude d'un résultat dépend de l'instant de sa livraison en plus de sa valeur.

Par ailleurs, l'évolution des exigences du contexte d'utilisation et la variation des contraintes de l'environnement d'exécution exhibent le besoin de développer des systèmes embarqués temps réel dynamiquement reconfigurables. Les reconfigurations dynamiques permettent de faire évoluer le système d'une configuration à une autre au cours de son exécution sans l'arrêter [50]. Ces reconfigurations peuvent être des reconfigurations architecturales ou comportementales. Les reconfigurations architecturales modifient la topologie d'une architecture en ajoutant ou en supprimant des connexions et/ou des composants tandis que les reconfigurations comportementales modifient le comportement du système en modifiant, par exemple, les propriétés, ainsi que le comportement, d'un composant.

Ces reconfigurations augmentent la complexité de développement des systèmes embarqués. De plus, elles peuvent provoquer le dysfonctionnement du système par la production d'erreurs et la perturbation de certaines propriétés non-fonctionnelles.

Ainsi, garantir le bon fonctionnement du système et respecter ses contraintes non-fonctionnelles sont devenus aujourd'hui des défis majeurs dans le développement des systèmes embarqués.

1.2 Problématique de la thèse

Dans ce travail de thèse, nous abordons le problème de construction d'un processus outillé pour le développement des systèmes temps réel répartis embarqués (que nous noterons TR^{2}E dans le reste du mémoire) dynamiquement reconfigurables. Ce processus doit répondre aux besoins de la modélisation de ces systèmes et en particulier des reconfigurations dynamiques à un haut niveau d'abstraction. Il doit permettre aussi la vérification de certaines propriétés non-fonctionnelles au niveau du modèle de conception ainsi que la génération de code à partir des modèles de haut niveau en se basant sur un intergiciel dédié pour des systèmes TR^{2}E dynamiquement reconfigurables.

1.2.1 Reconfiguration dynamique des systèmes TR^{2}E

Devant l'évolution rapide et la variation quotidienne des exigences des utilisateurs ainsi que la variation des contraintes de l'environnement d'exécution, les systèmes embarqués sont de plus en plus importants et demandés dans notre vie quotidienne. Cependant, la plupart des systèmes embarqués ne sont pas autonomes et exigent l'intervention humaine pour les reconfigurer. D'une part, l'intervention humaine peut causer des erreurs et exige beaucoup de temps et d'efforts. D'autre part, il est parfois impossible d'arrêter un système temps réel pour le reconfigurer. La reconfiguration dynamique est donc très importante pour les systèmes embarqués temps réel répartis.

En outre, les ressources matérielles d'un système embarqué sont généralement limitées et leur utilisation doit être optimisée. Afin de développer un système embarqué riche de plusieurs fonctionnalités et avec un faible coût en ressources matérielles, ces dernières doivent exécuter, à un instant donné, seulement les composants logiciels nécessaires. Elles doivent êtres allouées à la demande. Pour les architectures orientées composants, les composants doivent être remplacés et mis à jour au cours de l'exécution.

1.2.2 Développement des systèmes TR²E

Le développement des systèmes TR²E dynamiquement reconfigurables exige des efforts considérables et est source d'erreurs. Il est, en fait, très difficile et complexe de développer ces systèmes sans fournir un haut niveau d'abstraction. Pour ce faire, de nouveaux concepts de modélisation sont nécessaires pour spécifier ces systèmes et en particulier les reconfigurations dynamiques.

La plupart des travaux autour des systèmes reconfigurables proposent d'énumérer toutes les configurations possibles. Les travaux les plus connus ont été proposés dans le contexte des normes AADL [52] et MARTE [45]. Ces deux standards décrivent les reconfigurations dynamiques en termes de modes et de transitions entre eux. Un mode représente une configuration particulière tandis qu'une transition représente l'arrivée d'un événement qui contrôle et déclenche la reconfiguration d'un système d'un mode à un autre. Ces reconfigurations sont décrites en utilisant des machines à états qui présentent les modes possibles et les transitions entre eux. L'inconvénient majeur de ces deux standards est qu'ils nécessitent une énumération de toutes les configurations possibles du système. Or, cette énumération est la plupart du temps fastidieuse et assez prenante. De plus, elle peut provoquer l'oubli de certaines configurations.

Devant l'évolution croissante des exigences des systèmes embarqués, les développeurs doivent optimiser leurs temps de mise sur le marché (*time to market*) tout en fournissant des produits novateurs, concurrents et à faible coût. Pour cette raison, les développeurs doivent concevoir un système le plus rapidement possible en garantissant les performances exigées.

Afin de faire face à la complexité croissante de la modélisation des systèmes, plusieurs approches d'ingénierie ont été proposées. Les plus connues sont celles centrées sur les modèles et connues sous le nom "ingénierie dirigée par les modèles" (IDM) ou MDE (en ang. *Model Driven Engineering*) [53] et son standard de l'OMG : le MDA. En utilisant des langages de modélisation dans le MDE, les modèles représentent les artefacts principaux devant être construits et maintenus. Dans le contexte du MDE, le développement logiciel consiste à transformer un modèle en un autre plus raffiné jusqu'à obtenir un modèle final dédié à une plate-forme spécifique.

1.2.3 Vérification des systèmes TR²E

La reconfiguration dynamique complique considérablement le développement et l'exécution des systèmes TR²E. En effet, ces systèmes ont des contraintes temporelles

et de ressources strictes et exigent la préservation des propriétés non-fonctionnelles (telle que la consommation CPU) quand les reconfigurations s'appliquent. Une telle vérification des propriétés non-fonctionnelles au cours de l'exécution est très compliquée et inefficace pour deux raisons. Premièrement, elle nécessite des ressources matérielles supplémentaires qui sont déjà très limitées dans les systèmes embarqués. Deuxièmement, elle augmente le temps d'exécution ce qui peut dégrader le fonctionnement des systèmes critiques.

En outre, le coût de vérification et d'intégration est en croissance exponentielle. En effet, la majorité des projets [60] dépensent actuellement plus de 50% du coût de développement sur la vérification et l'intégration. Pour cela, la vérification des propriétés non-fonctionnelles des systèmes TR^{2}E reconfigurables doit être effectuée avant toute exécution. Cependant, la plupart des approches de vérification [10, 22, 49] considèrent seulement les systèmes statiques (non reconfigurables).

1.2.4 Intergiciels pour des systèmes TR^{2}E

Devant la complexité des systèmes embarqués dynamiquement reconfigurables, des intergiciels facilitant le développement de ces systèmes sont nécessaires. La plupart des intergiciels proposés supportant la reconfiguration dynamique ont plusieurs limites. Tout d'abord, à notre connaissance, il n'y a pas d'intergiciel qui assure la reconfiguration dynamique ainsi que la cohérence et la réflexivité des systèmes embarqués temps réel. La majorité des intergiciels existants [4, 58] assurent la reconfiguration dynamique mais sans garantir la cohérence et la réflexivité. Par conséquent, ces systèmes peuvent devenir incohérents après une reconfiguration. De plus, d'autres intergiciels assurant la reconfiguration dynamique [4, 26, 58] ne supportent pas les systèmes temps réel et ont des tailles de mémoire importantes. Enfin, les intergiciels assurant la reconfiguration dynamique ne couvrent pas tous les types de reconfiguration. Ces intergiciels supportent un ensemble limité de reconfigurations, architecturales [4, 26] ou comportementales [3].

1.3 Objectifs de la thèse

Pour remédier aux problèmes soulevés dans la section précédente, nous essayons dans cette thèse d'apporter des éléments de réponse pour améliorer la productivité des développeurs des systèmes TR^{2}E dynamiquement reconfigurables. Plus précisément, nous proposons de définir un processus de développement aidant les dévelop-

peurs à concevoir ces systèmes d'une manière plus facile et rapide tout en préservant les performances exigées. Il s'agit, en fait, de mettre en place les moyens et les outils (profils) nécessaires à la conception et au développement d'un système en passant d'un modèle à un haut niveau d'abstraction jusqu'à la génération de code. Pour cela, nous visons les contributions suivantes :

- Modélisation : Proposer de nouveaux concepts permettant de spécifier des systèmes TR^{2}E à un haut niveau d'abstraction et en particulier les reconfigurations dynamiques dans ces systèmes.
- Vérification : Permettre de vérifier certaines propriétés non-fonctionnelles au niveau modèle de conception pour aider au développement correct de ces systèmes.
- Plate-forme d'exécution : Développer un intergiciel dédié aux systèmes TR^{2}E dynamiquement reconfigurables avec une faible empreinte mémoire.
- Génération de code : Générer du code par des transformations successives à partir des modèles de haut niveau.

1.4 Solution proposée

Partant de l'objectif général de cette thèse, nous proposons une approche dirigée par les modèles [30] permettant de concevoir des systèmes TR^{2}E dynamiquement reconfigurables. En ce sens, nous définissons un processus de développement permettant de concevoir ces systèmes du niveau modèle jusqu'à la génération du code. Ce processus est censé aussi assurer la vérification de certaines propriétés non-fonctionnelles pour ces systèmes reconfigurables au niveau modèle. De plus, nous fournissons un intergiciel supportant la reconfiguration dynamique de ces systèmes et permettant la génération d'une grande partie du code.

1.4.1 Processus de développement des systèmes TR^{2}E

Afin de faciliter le développement des systèmes TR^{2}E dynamiquement reconfigurables, nous proposons une approche IDM. Cette approche permet la transformation d'un modèle en un autre modèle plus raffiné. Le passage d'un niveau à un autre se fait par ajout de détails jusqu'à l'obtention d'un modèle proche du code, ce qui facilite par la suite la génération du code.

Ce processus de développement permet à l'utilisateur de spécifier les reconfigurations dynamiques d'un système TR^{2}E ainsi que l'architecture logicielle du système.

Ensuite, il permet de spécifier la partie matérielle suivie de l'allocation de l'architecture logicielle sur une instance de l'architecture matérielle. Finalement, tous ces modèles sont par la suite transformés en un modèle représentant le code généré.

1.4.2 Modélisation des systèmes TR²E

Contrairement à AADL [52] et MARTE [45], nous proposons une approche basée sur les modèles pour la spécification des systèmes TR²E dynamiquement reconfigurables sans énumérer toutes les configurations possibles. Pour cela, nous introduisons la notion de *MetaMode* [32] qui capture et caractérise un ensemble de configurations au lieu de définir chacune d'entre elles. Le *MetaMode* est défini par un ensemble de composants structurés et des connexions potentielles entre ces composants ainsi que des contraintes structurelles et non-fonctionnelles. Les reconfigurations dynamiques sont spécifiées via des machines à états composées par des *MetaModes* et des transitions entre eux.

1.4.3 Vérification des systèmes TR²E

Afin d'assurer la construction correcte des systèmes TR²E dynamiquement reconfigurables, notre approche permet de vérifier certaines propriétés non-fonctionnelles au niveau modèle de conception. Elle permet de vérifier la consommation CPU, le respect des échéances des tâches, la consommation mémoire, la consommation de la bande passante, l'absence d'interblocage et l'absence de famine. Si l'une de ces propriétés n'est pas satisfaite, les modèles spécifiant le système devront être rectifiés.

1.4.4 Intergiciel dédié aux systèmes TR²E

Une fois les phases de modélisation et de vérification accomplies, nous continuons par la proposition d'un intergiciel dédié aux systèmes TR²E dynamiquement reconfigurables. Cet intergiciel aide à la génération d'une grande partie du code de ces systèmes. En effet, il offre des routines assurant la reconfiguration dynamique ainsi que la supervision et la cohérence. Le déterminisme de ses fonctionnalités le rend particulièrement adéquat pour les systèmes temps réel et sa faible empreinte mémoire permet de l'utiliser pour les systèmes embarqués.

1.5 Plan du mémoire

Ce document est organisé en huit chapitres, le premier étant cette introduction générale présentant le contexte, le problématique et les principales contributions de la thèse. Le chapitre 2 est consacré à un état de l'art dédié à la description et l'analyse des processus et des outils liés au développement des systèmes embarqués. La première partie montre des outils et des langages de modélisation des systèmes embarqués. Ensuite, les algorithmes d'ordonnancement assurant l'ordonnancement des systèmes embarqués et des frameworks de vérification à base de modèles des propriétés non-fonctionnelles des systèmes embarqués sont décrits. Des intergiciels dédiés aux systèmes embarqués ont été aussi étudiés. Enfin, une étude des différents processus de développement des systèmes embarqués est proposée.

Le chapitre 3 présente un processus de développement pour les systèmes TR^{2}E dynamiquement reconfigurables avec ses différentes phases. La première phase du processus, celle de modélisation, est aussi décrite dans ce chapitre. Les deux phases de vérification et de génération sont décrites dans les deux chapitres 4 et 5. Le chapitre 4 présente la phase de vérification du processus de développement. Il présente la vérification de certaines propriétés (consommation CPU, respect des échéances des tâches, consommation mémoire, consommation de la bande passante, absence d'interblocage et absence de famine) au niveau modèle de conception. Le chapitre 5 présente la phase de génération du processus. La première partie du chapitre 5 est consacrée à décrire l'intergiciel proposé et dédié aux systèmes TR^{2}E dynamiquement reconfigurables. Tandis que la deuxième partie présente la génération du code et la configuration automatique de l'intergiciel.

Afin de mettre en oeuvre notre approche, le chapitre 6 présente l'outillage du processus en mettant l'accent sur les outils et les profils introduits pour favoriser son utilisation. Il s'agit d'une chaîne d'outils mettant en oeuvre le processus du développement des systèmes TR^{2}E dynamiquement reconfigurables proposé. Ces outils permettent la modélisation, la vérification et la génération du code des systèmes. Une illustration du processus proposé par l'étude de cas du système de localisation *MyGPS* est présentée dans le chapitre 7. Toutes les étapes du processus de développement proposé sont illustrées par cette étude de cas.

Finalement, le chapitre 8 conclut ce document en rappelant les contributions principales de la thèse. Ces contributions aboutissent à d'autres axes de développement intéressants présentant les perspectives de cette thèse.

Chapitre 2

État de l'art

2.1 Introduction

L'évolution des exigences du contexte d'utilisation et la variation des contraintes de l'environnement d'exécution motivent le besoin de développer des systèmes embarqués temps réel dynamiquement reconfigurables. Les reconfigurations dynamiques entraînent des modifications architecturales ou comportementales d'une architecture logicielle en cours d'exécution. Ces reconfigurations augmentent la complexité de développement des systèmes TR^{2}E. De plus, elles peuvent provoquer le dysfonctionnement du système par la production d'erreurs et la perturbation de certaines propriétés non-fonctionnelles. Ainsi, concevoir des systèmes TR^{2}E dynamiquement reconfigurables et garantir leur bon fonctionnement, en respectant des contraintes non-fonctionnelles, sont devenus aujourd'hui des défis dans le développement des logiciels.

Dans ce chapitre, nous présentons un état de l'art autour du développement des systèmes embarqués et plus particulièrement les systèmes dynamiquement reconfigurables, en mettant en évidence les critères de comparaison les plus importants entre les approches étudiées. Dans ce contexte, un état de l'art bien détaillé a été publié dans [27]. Dans un premier temps, nous focalisons notre étude sur les outils et les langages de modélisation des systèmes embarqués dynamiquement reconfigurables. Puis, nous présentons les algorithmes d'ordonnancement les plus connus. Une synthèse des divers travaux qui s'articulent autour de la vérification de certaines propriétés non-fonctionnelles au niveau modèle de conception est ensuite effectuée. Cette étude nous a permis par la suite d'analyser les principaux intergiciels dédiés aux systèmes embarqués. Pour traiter l'aspect méthodologique, nous étudions les

processus de développement permettant de concevoir des systèmes embarqués dynamiquement reconfigurables. Enfin, nous terminons ce chapitre par une conclusion.

2.2 Modélisation des systèmes embarqués

Dans cette section, nous présentons les outils et les langages permettant de spécifier des systèmes embarqués temps réel et en particulier les systèmes reconfigurables.

2.2.1 AADL

AADL (Architecture Analysis & Design Language) [52] est un langage de description d'architecture permettant la spécification des systèmes TR^{2}E. Il est un standard international publié par la SAE (Society of Automotive Engineers).

AADL repose principalement sur la notion de "composant". Il permet de décrire des systèmes embarqués temps réel en assemblant des blocs développés séparément. Il définit une interface précise pour chaque composant et il sépare l'implantation interne du composant de la description de l'interface. Un composant représente une entité matérielle ou logicielle qui appartient au système. AADL permet aussi de décrire à la fois la partie matérielle et la partie logicielle d'un système. Pour cela, il utilise une syntaxe textuelle aussi bien qu'une représentation graphique.

AADL permet aussi de spécifier les reconfigurations dynamiques des systèmes TR^{2}E via des machines à états composées par des modes et des transitions entre eux. Seuls les événements (event ports) définis au niveau des composants peuvent déclencher un changement de mode. Ces concepts permettent de modéliser les reconfigurations d'un composant en fonction d'événements. La transition d'un mode à un autre nécessite des activations et/ou désactivations de certains sous-composants, connexions, etc.

Discussion :

AADL est un langage standard défini afin de spécifier des systèmes TR^{2}E. En effet, il permet de spécifier la partie logicielle ainsi que la partie matérielle de ces systèmes par des composants qui peuvent être logiciels ou matériels. AADL permet aussi de spécifier les reconfigurations de ces systèmes par des machines à états composées par des modes et des transitions entre eux. Cependant, il faut énumérer tous les modes possibles pour spécifier les reconfigurations dynamiques par des transitions entre ces modes. Et cela représente une tâche fastidieuse dans le cas de systèmes ayant un grand nombre de modes. Il y a aussi le risque de ne pas couvrir

tous les modes possibles du système. De plus, AADL permet de spécifier un système embarqué à un niveau concret (threads, processeurs, etc), donc la modélisation des reconfigurations est liée à une plate-forme spécifique.

2.2.2 Profil MARTE

MARTE (Modeling and Analysis of Real-Time Embedded systems) [45] est un profil UML standard de l'OMG, inspiré du profil SPT [42] et assurant la modélisation et l'analyse des systèmes embarqués temps réel. Il permet la modélisation des systèmes embarqués temps réel à plusieurs niveaux d'abstraction grâce aux différents paquetages (sous profils) offerts. Ces paquetages permettent la séparation entre les deux parties, logicielle et matérielle, du système. En outre, MARTE aide à la spécification des propriétés non-fonctionnelles comme le temps et l'empreinte mémoire et les contraintes temporelles complexes. Pour cela, il offre le langage VSL (Value Specification Language) sous la forme d'une extension du langage déclaratif OCL afin de supporter les contraintes comportementales. Il offre aussi une bibliothèque définissant de nouveaux types utiles pour la spécification des propriétés non-fonctionnelles.

Par ailleurs, MARTE permet de spécifier les reconfigurations comportementales des systèmes embarqués temps réel en utilisant une machine à états composée par des modes et des transitions entre eux. Un mode représente un état particulier du système tandis qu'une transition représente un événement qui déclenche une reconfiguration. La figure 2.1 présente le paquetage *CommonBehavior* du profil MARTE définissant les reconfigurations comportementales. Chaque configuration du système a plusieurs modes définissant des comportements différents. MARTE décrit les systèmes reconfigurables par des modes opérationnels. La transition d'un mode à un autre se fait par des reconfigurations modifiant les caractéristiques internes des composants. Les transitions sont déclenchées par des événements. Le diagramme de collaboration est utilisé pour représenter un mode opérationnel tandis que le diagramme d'états-transitions est utilisé pour représenter les transitions entre les modes.

Discussion :

Le profil MARTE étend le langage UML par l'ajout de nouveaux concepts permettant la modélisation et l'analyse des systèmes embarqués temps réel.

Mais il n'assure que la modélisation des reconfigurations comportementales du système. Il ne prend pas en considération d'autres types de reconfiguration comme les reconfigurations architecturales. MARTE spécifie les reconfigurations dynamiques par l'énumération de l'ensemble des modes possibles du système. Ceci paraît fasti-

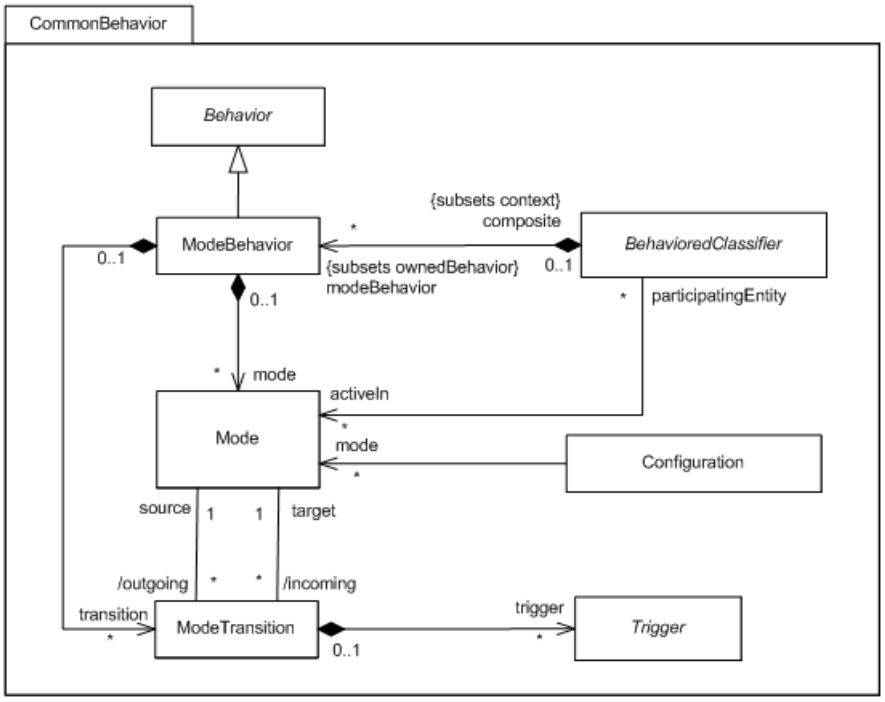

FIGURE 2.1 – Paquetage *CommonBehavior* du profil MARTE [45]

dieux particulièrement pour les systèmes pouvant avoir un grand nombre de modes. De plus, il y a le risque de ne pas couvrir tous les modes possibles du système.

2.2.3 Spécification PEARL et le profil UML-RT

La spécification PEARL [16] a été proposée pour assurer la programmation des applications temps réel automatiques. PEARL est étendue par des constructions permettant la description des configurations matérielles et logicielles. Elle définit aussi le mapping des modules logiciels sur des composants matériels. Elle introduit des attributs définissant des informations sur le temps permettant de spécifier des systèmes temps réel. Elle permet de spécifier des communications avec leurs caractéristiques ainsi que des architectures tolérantes aux pannes. De plus, PEARL permet la modélisation graphique. Elle assure aussi la spécification des conditions et des méthodes assurant des reconfigurations dynamiques en énumérant toutes les configurations possibles.

En se basant sur les concepts introduits dans la spécification PEARL, un patron de reconfiguration UML pour des systèmes embarqués [17] a été proposé. Ce modèle de reconfiguration est décrit par un profil *UML-RT* permettant d'assurer la modé-

lisation et la gestion des reconfigurations des architectures logicielles et matérielles. *UML-RT* utilise le diagramme de séquence pour modéliser les différentes reconfigurations et plus précisément les transactions d'une configuration à une autre. En UML, le gestionnaire de reconfiguration est modélisé selon les trois aspects suivants :

- L'aspect comportemental qui présente les modes du système et les conditions sous lesquelles le système change de mode,
- L'aspect structurel qui définit trois niveaux d'abstraction : le niveau station, le niveau configuration et le niveau collection,
- L'aspect fonctionnel qui fournit les fonctionnalités définissant les communications et les interfaces de l'application.

Afin d'assurer la sécurité des reconfigurations modélisées par le profil UML-RT, des fonctionnalités de contrôle [18] ont été implantées. Ces fonctionnalités assurent la vérification des contraintes d'entrées/sorties, des contraintes des états (configurations) et des contraintes temporelles ainsi que le traitement des exceptions.

Discussion :

La spécification PEARL a été proposée pour aider à la programmation des applications temps réel automatiques. Elle définit un système embarqué temps réel par l'allocation d'une configuration logicielle sur une configuration matérielle. Le profil *UML-RT* de cette spécification permet de spécifier les reconfigurations dynamiques de ces systèmes par des transactions entre un ensemble de configurations.

Cependant, les concepts définis par la spécification PEARL restent limités. Ils ne permettent ni de spécifier des propriétés et des contraintes non-fonctionnelles de ces systèmes ni de spécifier des contraintes temporelles complexes. De plus, la spécification des reconfigurations consiste à énumérer toutes les configurations possibles.

2.2.4 FRACTAL ADL et langage de domaine FScript

FRACTAL ADL [51] est un langage de description d'architecture utilisé pour spécifier des architectures à base de composants FRACTAL. Le modèle de composant FRACTAL permet la modélisation et le déploiement des systèmes logiciels complexes tel que les intergiciels et les systèmes d'exploitation. FRACTAL ADL est un langage déclaratif de haut niveau qui permet la description des configurations logicielles en terme de composants composites avec ses interfaces, attributs et connexions.

Le modèle de composant FRACTAL est défini en terme d'APIs qui permettent de dévoiler la structure d'une application FRACTAL et de la reconfigurer au cours de l'exécution. La programmation des reconfigurations dynamiques directement à ce

niveau rend le code difficilement compréhensible. Il est donc nécessaire de faire des analyses pour garantir le comportement avant l'exécution d'une reconfiguration, ce qui peut être essentiel pour les systèmes critiques.

FScript [50] est un langage spécifique de domaine (*Domain Specific Language*) défini pour surmonter ces limitations tout en conservant les avantages de FRACTAL. Il permet de spécifier les actions de reconfiguration. Il peut être utilisé pour naviguer simplement dans une architecture FRACTAL et sélectionner des parties de cette architecture sur lesquelles les reconfigurations peuvent être ensuite appliquées.

L'approche proposée dans [50] assure la reconfiguration dynamique des systèmes embarqués avec des ressources limitées. Pour cela, il relie le framework basé composant THINK pour construire des systèmes d'exploitation et le langage FScript pour présenter les reconfigurations. En effet, Think est un framework général pour la construction des systèmes basés composant et spécialement des noyaux de systèmes d'exploitation. Ce framework contient une implantation en C du modèle de composant FRACTAL, un ADL (FRACTAL ADL) avec le compilateur associé "ADL-to-C" et une bibliothèque de composants appelée "Kortex".

Fscript a des avantages comme la puissance d'expression, la vérification que l'algorithme de reconfiguration soit conforme à l'architecture cible et l'évolution facile de l'architecture du noyau par l'adaptation et la maintenance de l'algorithme de reconfiguration.

Discussion :

FScript et le framework Think se rejoignent pour développer des systèmes d'exploitation reconfigurables. Cette approche satisfait les objectifs de flexibilité, sécurité et simplicité mais elle ne couvre pas les objectifs d'efficacité. De plus, FScript ne permet pas de spécifier des actions de reconfiguration à un haut niveau d'abstraction. Il présente les actions de reconfiguration à un niveau concret par l'implantation des actions de reconfiguration reliées à une plate-forme spécifique. Cette approche ne peut être utilisée que pour des applications orientées composants et seulement avec le modèle de composant FRACTAL.

2.2.5 Synthèse

Les deux standards AADL [52] et MARTE [45] permettent la spécification des systèmes embarqués temps réels avec leurs propriétés non-fonctionnelles et leurs contraintes temporelles. Ils décrivent les reconfigurations dynamiques des systèmes par des machines à états composées par des modes et des transitions entre eux. Dans

AADL, un mode représente une configuration particulière. Par contre, pour le profil MARTE, un mode présente un comportement d'une configuration donnée.

D'une part, [14] montre comment le profil MARTE peut représenter les modèles AADL des systèmes temps réel embarqués. D'autre part, [52] déclare que le profil UML qui supporte la modélisation avec des concepts AADL est le profil MARTE de l'OMG. Une annexe dans le profil MARTE [45] a été introduite afin de montrer la correspondance entre le profil MARTE et quelques concepts de AADL.

Par contre et contrairement à AADL, le profil MARTE traite seulement les reconfigurations comportementales des systèmes embarqués temps réel par la modification des caractéristiques internes des composants. De plus, AADL spécifie les systèmes embarqués à un niveau concret (threads, processeurs, etc) lié à une plate-forme spécifique.

De même, la spécification PEARL [17] permet aussi la description des systèmes embarqués temps réel. Elle assure la transaction d'une configuration à une autre par un ensemble d'actions de reconfiguration. Ces transactions sont déclenchées par des événements. La spécification PEARL prend en considération la tolérance aux pannes qui nécessite une architecture reconfigurable. *UML-RT* est un profil qui définit les concepts de la spécification PEARL. Cependant, la spécification PEARL reste très limitée par rapport à AADL et MARTE pour la spécification des systèmes embarqués temps réel. Elle ne permet pas de spécifier les propriétés et les contraintes non-fonctionnelles ainsi que les expressions temporelles complexes. De plus, PEARL impose de spécifier des systèmes dynamiquement reconfigurables en énumérant toutes les configurations possibles comme AADL et MARTE. Cette énumération présente un inconvénient majeur pour des systèmes ayant un grand nombre de configurations, car elle peut s'avérer pénible et consommatrice de temps. De même, il y a le risque de ne pas couvrir toutes les configurations ou toutes les transitions possibles.

Les langages de domaine peuvent aider à la spécification des systèmes TR^{2}E reconfigurables comme celui de FScript [50]. Mais ces langages exigent des compétences de la part du développeur surtout qu'ils permettent de spécifier des actions de reconfiguration à un niveau concret.

Notre objectif est donc de proposer une approche dirigée par les modèles introduisant de nouveaux concepts permettant de spécifier des systèmes TR^{2}E dynamiquement reconfigurables à un haut niveau d'abstraction sans énumération de toutes les configurations possibles. Cette approche doit prendre en considération les propriétés fonctionnelles et non-fonctionnelles de ces systèmes. Nous avons aussi besoin

de vérifier certaines propriétés non-fonctionnelles au niveau modèle de conception. Pour cela, nous pouvons utiliser les algorithmes d'ordonnancement pour vérifier la consommation CPU et le respect des échéances des tâches. Dans ce qui suit, nous décrivons les algorithmes d'ordonnancement.

2.3 Algorithmes d'ordonnancement

Les algorithmes d'ordonnancement [39] permettent de vérifier des contraintes temporelles (e.g. respect des échéances des tâches) et des contraintes de ressources (e.g. consommation CPU). Un algorithme d'ordonnancement représente une politique d'exécution des tâches périodiques sur un processeur. Il aide à garantir l'exécution des tâches sur un processeur tout en assurant le respect de leurs échéances et à vérifier la consommation du processeur.

Les politiques d'ordonnancement temps réel sont de deux types : l'ordonnancement par priorité statique et l'ordonnancement par priorité dynamique. Dans la suite, nous présentons un aperçu de ces deux types d'ordonnancement ainsi que les algorithmes les plus utilisés.

2.3.1 Ordonnancement par priorité statique

L'ordonnancement par priorité statique consiste à fixer les priorités des tâches avant de commencer leur exécution. Ces priorités seront préservées tout au long de l'exécution. Parmi les algorithmes d'ordonnancement par priorité statique, nous citons les deux algorithmes RMS et DMS.

Pour RMS (*Rate Monotonic Scheduling*) [39], la tâche ayant la période la plus petite a la priorité la plus grande. Donc, les priorités des différentes tâches seront fixées dès le départ.

Dans RMS, l'ordonnancement des tâches sur un processeur est garanti si la condition 2.1 est vérifiée c'est à dire si le taux d'utilisation du processeur est inférieur à un seuil défini en fonction du nombre des tâches périodiques. Cette condition est suffisante. Donc les tâches peuvent parfois être ordonnancées même si la condition suffisante n'est pas satisfaite.

$$\sum_{i=0}^{n}(C_i/P_i) \leq n(2^{1/n} - 1) \tag{2.1}$$

n : nombre de tâches périodiques,

C_i : pire temps d'exécution de la tâche i,

P_i : période de la tâche i.

Pour DMS (*Deadline Monotonic Scheduling*) [38], la tâche ayant l'échéance la plus petite a la priorité la plus grande. Il a la même condition d'ordonnançabilité que celui de RMS.

2.3.2 Ordonnancement par priorité dynamique

L'ordonnancement par priorité dynamique consiste à modifier les priorités des tâches durant leur exécution. Parmi les algorithmes d'ordonnancement par priorité dynamique, nous citons les deux algorithmes EDF et LLF.

Pour EDF (*Earliest Deadline First*) [39], la tâche ayant l'échéance la plus proche est la tâche la plus prioritaire. Les priorités des tâches changent généralement au cours de l'exécution du système puisque les temps les séparant de leurs échéances peuvent changer.

Dans EDF, les tâches ne sont ordonnançables que si la condition 2.2 est vérifiée : le processeur ne doit pas être surchargé.

$$\sum_{i=0}^{n}(C_i/P_i) \leq 1 \tag{2.2}$$

n : nombre de tâches périodiques,

C_i : pire temps d'exécution de la tâche i,

P_i : période de la tâche i.

Pour LLF (*Least Laxity First*) [24], la tâche ayant la marge (*laxity*) la plus petite est la plus prioritaire. La marge L est définie par la formule 2.3.

$$L(t) = D - t - T_r(t) \tag{2.3}$$

L : marge de la tâche

D : échéance de la tâche.

T_r : temps d'exécution restant de la tâche.

t : temps courant du système.

Dans LLF, la condition d'ordonnançabilité est la même que celle de EDF.

2.3.3 Synthèse

Le tableau 2.1 présente une comparaison entre des algorithmes d'ordonnancement à priorité dynamique et ceux à priorité statique. Pour les algorithmes à priorité dynamique (EDF et LLF), la consommation CPU peut atteindre 100 %. En effet, les deux algorithmes EDF et LLF déterminent à chaque instant au cours de l'exécution respectivement l'échéance des tâches la plus proche et la laxité des tâches la plus petite. Par conséquent, la mise en oeuvre de ces deux algorithmes, et en particulier LLF, sera très complexe. De plus, un comportement supplémentaire du système sera généré (*overhead*) nécessitant donc plus de consommation de ressources (consommation CPU) et plus de temps d'exécution. Cela limite l'utilisation de deux algorithmes EDF et LLF.

Contrairement aux algorithmes à priorité dynamique, les algorithmes à priorité statique se comportent mieux. Ils ne conduisent pas à un comportement supplémentaire. De plus, RMS et DMS se comportent mieux par rapport à EDF et LLF en cas de surcharge, et ils s'implantent facilement avec les systèmes d'exploitation classiques. En effet, RMS et DMS sont optimaux pour les ordonnancements à priorité statique. Avec leur implantation simple, ils sont donc considérés les meilleurs.

TABLE 2.1 – Tableau comparatif des algorithmes d'ordonnancement

caractéristiques	**RMS**	**DMS**	**EDF**	**LLF**
Priorité	statique	statique	dynamique	dynamique
Utilisation du CPU	$\leq 69\%$ qd $n \to \infty$	$\leq 69\%$ qd $n \to \infty$	$\leq 100\%$	$\leq 100\%$
Simplicité de mise en oeuvre	✓	✓	✘	✘
Optimalité	✓	✓	✓	✓
Meilleur en cas de surcharge	✓	✓	✘	✘
Répandu avec les exécutifs classiques	✓	✓	✘	

Dans ce qui suit, nous présentons des travaux qui s'articulent autour de la vérification de certaines propriétés non-fonctionnelles au niveau modèle de conception.

2.4 Frameworks de vérification

Plusieurs frameworks assurant la vérification à base de modèles pour des systèmes embarqués ont été proposés. Dans ce qui suit, nous présentons des frameworks assurant la vérification des propriétés temporelles et/ou des propriétés liées

aux ressources.

2.4.1 Cheddar

Cheddar [56] est un framework écrit en Ada et fournissant des outils pour vérifier des critères de performance. Il vérifie, par exemple, si une application temps réel respecte ses contraintes temporelles. Cheddar assure les tests de faisabilité et la simulation des systèmes mono-processeurs ou multi-processeurs. Les tests de faisabilité consistent à :

- comparer le facteur d'utilisation d'un processeur avec une borne donnée,
- comparer le pire temps d'exécution de chaque tâche avec son échéance,
- trouver une borne pour le facteur d'utilisation du tampon dans le cas d'un tampon partagé entre des tâches périodiques ordonnancées par un ordonnanceur de priorité fixe ou de priorité dynamique.

La simulation consiste à simuler l'exécution du système et à effectuer l'analyse d'ordonnancement pour vérifier certaines propriétés. A partir de l'analyse d'ordonnancement, ce framework permet d'obtenir les temps de réponse des tâches, les temps de blocage sur des ressources partagées, le nombre de préemptions et les échéances non respectées. De plus, il permet de détecter les inter-blocages et les inversions de priorité. En effet, ce framework supporte la plupart des algorithmes d'ordonnancement (RMS, DMS, EDF et LLF).

Cheddar est un framework ouvert et flexible. En effet, il interagit facilement avec des outils CASE par l'envoi et la réception des fichiers XML. De plus, il peut être étendu afin d'exécuter des ordonnancements spécifiques, faire des analyses spécifiques, ou ordonnancer des tâches avec des modèles particuliers d'activation.

Discussion :

Les ordonnanceurs, les modèles d'activation des tâches et les analyseurs d'événements exprimés avec le langage Ada seront interprétés lors d'une simulation et non pas compilés par le framework. Il est donc facile d'écrire et de tester des extensions sur ce framework sans une connaissance approfondie du langage Ada.

Cependant, Cheddar vérifie certaines propriétés non-fonctionnelles pour des systèmes ayant seulement des tâches périodiques. Il ne vérifie pas les systèmes ayant des tâches sporadiques et/ou apériodiques. Cheddar ne permet pas non plus de vérifier certaines propriétés liées aux ressources comme la consommation mémoire et la consommation de la bande passante.

2.4.2 TASM Toolset

TASM Toolset [48, 49] implante les caractéristiques du langage TASM à travers trois composants principaux :

- Un éditeur permettant l'édition du texte de base et la représentation syntaxique des fonctionnalités en utilisant le langage TASM.
- Un simulateur permettant la visualisation du comportement dynamique des spécifications TASM y compris les dépendances du temps et la consommation des ressources.
- Un analyseur assurant la vérification des propriétés de base des spécifications TASM comme la consistance.

Le langage TASM vise à capturer trois aspects du comportement d'un système temps réel : le comportement fonctionnel, le comportement temporel, et la consommation d'énergie.

Afin de vérifier les propriétés temporelles, TASM Toolset intègre le vérificateur de modèle UPPAAL [2]. Pour cela, il effectue une traduction de la spécification TASM vers les automates temporisés. En vue de vérifier la consistance de la spécification, TASM Toolset intègre le solveur SAT4J SAT pour vérifier une formule booléenne obtenue à partir des règles sur des machines à états.

Discussion :

TASM Toolset fournit une approche basée sur la spécification pour développer des systèmes embarqués temps réel. Il permet de spécifier, analyser et tester les trois aspects du comportement de ces systèmes (fonctionnel, temporel et consommation d'énergie). Cependant, TASM Toolset ne permet pas de vérifier les propriétés liées aux ressources comme la consommation CPU, la consommation mémoire et la consommation de la bande passante. En outre, la spécification en utilisant le langage TASM est compliquée et elle exige des compétences supplémentaires.

2.4.3 ModSyn

ModSyn (*Model driven Co-synthesis for embedded system*) [10] est un framework basé sur l'ingénierie dirigée par les modèles (IDM) assurant la vérification formelle des systèmes embarqués. Il permet de vérifier des propriétés logiques fonctionnelles et temporelles et de vérifier aussi l'exécution correcte de ces systèmes.

Ce framework permet de générer automatiquement un réseau d'automates temporisés à partir des modèles UML présentant la spécification fonctionnelle d'une application embarquée par un diagramme de classes et des diagrammes de séquence.

En effet, les modèles UML sont transformés vers un modèle commun d'application défini par le méta-modèle IAMM (Internal Application Meta-model) proposé. Ensuite, ce modèle commun d'application est transformé vers un modèle de réseau d'automates temporisés conforme au méta-modèle LTAMM (Label Timed Automata Meta-model) proposé. La transformation entre les modèles est implantée avec le langage *Xtend* du framework *Open ArchitectureWare*.

Afin de vérifier certaines propriétés temporelles et fonctionnelles du système, une représentation du réseau d'automates temporisés est obtenue automatiquement comme une entrée pour des outils de vérification formelle comme UPPAAL [2].

Discussion :

ModSyn est un framework assurant la vérification formelle de certaines propriétés temporelles et fonctionnelles des systèmes embarqués temps réel en se basant sur une approche fondée sur l'ingénierie dirigée par les modèles. Cependant, ce framework ne supporte ni les systèmes répartis ni les systèmes dynamiquement reconfigurables. De plus, il ne vérifie pas les propriétés liées aux ressources comme la consommation CPU, la consommation mémoire et la consommation de la bande passante des bus.

2.4.4 Dream

Dream (Distributed Realtime Embedded Analysis Method) [40] est un framework assurant la vérification des systèmes TR^{2}E. Il vérifie les propriétés principales de QoS (*Quality of service*) des systèmes TR^{2}E à base de composants utilisant le modèle de communication de publication/souscription de messages (publish/subscribe communication). Ce framework permet la simulation et la vérification des systèmes TR^{2}E en utilisant le formalisme des automates temporisés et un vérificateur de modèles comme par exemple UPPAAL [2]. Il permet donc d'assurer l'analyse d'ordonnancement et la vérification des propriétés de QoS.

Dream utilise un modèle de domaine sémantique des systèmes TR^{2}E (DRE Semantic Domain) pour vérifier l'ordonnancement non préemptif des applications avioniques sur la plate-forme *Boeing Bold Stroke*. Cette plate-forme inclut un intergiciel développé en utilisant RT-CORBA [41]. Ce modèle de domaine contient des automates temporisés spécifiant les composants de base pour des systèmes TR^{2}E comme les timers, les ordonnanceurs et les tâches. Dream supporte une transformation de graphe spécifiée avec le modèle de transformation GREAT qui prend comme des entrées les modèles de Boeing Bold Stroke et génère des automates temporisés représentant le système.

Discussion :

Dream est un framework assurant la vérification des propriétés de QoS des systèmes TR^{2}E. Il permet aussi de vérifier l'absence d'interblocage, l'ordonnancement du système et la consommation mémoire. Cependant, Dream ne vérifie pas la bande passante des bus bien qu'il couvre les systèmes répartis. Il ne vérifie pas aussi la consommation CPU. En outre, ce framework ne prend pas en considération les systèmes dynamiquement reconfigurables.

2.4.5 VERTAF

VERTAF (*Verifiable Embedded Real-Time Application Framework*) [22] est un framework supportant la vérification des systèmes embarqués temps réel. Il intègre trois techniques : la réutilisation basée composant, la synthèse formelle (ordonnancement, génération de code), et la vérification formelle. Il assure la génération de code à partir des modèles UML. La phase de conception utilise trois diagrammes UML : le diagramme de classes, le diagramme d'états-transitions et le diagramme de séquence.

Afin d'assurer l'ordonnancement, les diagrammes d'UML sont traduits vers des réseaux de Petri RTPN (*Real-Time Petri Nets*) ou CCPN (*Complexe Choice Petri Nets*). A partir des diagrammes de séquence, les modèles RTPN et CCPN sont automatiquement générés. Pour vérifier l'ordonnancement, VERTAF a recours à deux algorithmes d'ordonnancement :

- Si le système est temps réel, un ordonnancement quasi dynamique (QDS) est appliqué sur des modèles RTPN spécifiant le système. Il prépare le système pour être généré comme un seul noyau exécutif temps réel avec un ordonnanceur.
- Si le système n'est pas temps réel, un ordonnancement quasi statique étendu (EQSS) est appliqué sur des modèles CCPN spécifiant le système. Il prépare le système pour être généré comme un ensemble de tâches qui peuvent être ordonnancées et envoyées par un RTOS comme MicroC/II OS ou Linux ARM.

Pour assurer la vérification formelle, des automates temporisés étendus ETAs seront générés à partir des diagrammes d'états-transitions. Afin de vérifier certaines propriétés, ces automates sont des entrées pour le vérificateur de modèles SGM (*State Graph Manipulators*). VERTAF vérifie deux classes de propriétés : des propriétés du système comme l'absence d'interblocage et la vivacité et des propriétés définies par l'utilisateur à partir de la spécification UML en utilisant le langage déclaratif OCL.

Toutes ces propriétés sont automatiquement traduites vers des spécifications TCTL pour être vérifiées par SGM.

Discussion :

VERTAF est un framework désigné pour le développement des systèmes embarqués temps réel. Il repose sur la réutilisation des composants logiciels, la synthèse formelle et la vérification formelle. VERTAF assure la modélisation de ces systèmes par des diagrammes UML. Ces diagrammes sont traduits vers des formalismes formels (automates temporisés étendus, RTPN, CCPN) afin de vérifier certaines propriétés du système.

VERTAF modélise les systèmes embarqués temps réel seulement par des diagrammes UML. Cependant, UML est un langage générique et ne permet pas de spécifier en détail ces systèmes. Bien que VERTAF vérifie certaines propriétés du système, il n'assure pas la vérification des propriétés liées aux ressources comme par exemple la consommation mémoire, la consommation CPU ou la consommation de la bande passante. En plus, ce framework ne supporte pas les systèmes TR^{2}E dynamiquement reconfigurables et il n'assure ni la modélisation ni la vérification formelle de ces systèmes.

2.4.6 Synthèse

Afin de garantir le bon fonctionnement des systèmes TR^{2}E, nous nous intéressons à la vérification basée modèle des propriétés non-fonctionnelles des systèmes TR^{2}E dynamiquement reconfigurables.

Dans la littérature, plusieurs frameworks de vérification à base de modèles ont été proposés. Ils permettent la vérification des propriétés non-fonctionnelles comme les propriétés temporelles et les propriétés liées aux ressources. Ces frameworks utilisent différents langages formels et outils de vérification. Cependant, la plupart de ces frameworks ne supportent ni les systèmes dynamiquement reconfigurables ni les systèmes répartis comme le montre le tableau 2.2.

En vue de répondre aux exigences des utilisateurs et aux changements de l'environnement, la reconfiguration dynamique devient de plus en plus importante. Toutefois, la reconfiguration au cours de l'exécution peut affecter l'exécution du système et provoquer le dysfonctionnement par la production d'anomalies et la violation des propriétés non-fonctionnelles. Il est donc nécessaire de vérifier certaines propriétés non-fonctionnelles pour des systèmes embarqués temps réel dynamiquement reconfigurables. Devant les contraintes temporelles et la limitation des ressources des sys-

TABLE 2.2 – Tableau récapitulatif des approches et leurs domaines d'utilisation

Frameworks	Systèmes embarqués temps réel	Systèmes répartis	Systèmes reconfigurables
VERTAF [22]	✓	✘	✘
Cheddar [56]	✓	✘	✘
Dream [40]	✓	✓	✘
Tasm Toolset [48, 49]	✓	✘	✘
ModSyn [10]	✓	✘	✘

tèmes embarqués temps réel, nous visons à vérifier la préservation de ces propriétés au niveau modèle et non pas au cours de l'exécution. Plus précisément, nous visons à vérifier la consommation CPU, la consommation mémoire, la consommation de la bande passante, le respect des échéances des tâches et l'absence d'interblocage et de famine.

Par ailleurs, la plupart des travaux existants permettent de vérifier des propriétés non-fonctionnelles pour des systèmes statiques et n'assurent pas la vérification de toutes les propriétés que nous visons. Le tableau 2.3 présente une comparaison de frameworks vérifiant des propriétés non-fonctionnelles pour des systèmes embarqués au niveau modèle.

TASM Toolset [49] permet de vérifier des propriétés liées aux ressources (consommation mémoire, consommation CPU et consommation de la bande passante) ainsi que l'absence d'interblocage en utilisant le langage TASM. Mais il ne permet pas de vérifier le respect des échéances et l'absence de famine. Le respect des échéances peut être vérifié par les automates temporisés [22, 40] ou par les algorithmes d'ordonnancement [56]. Nous pouvons vérifier seulement le respect des échéances des tâches non préemptives en utilisant les automates temporisés.

L'absence d'interblocage et l'absence de famine sont, quant à elles, généralement vérifiées en utilisant les automates temporisés et les vérificateurs de modèles [10, 22, 40, 49]. Par contre, les automates temporisés sont limités par le problème d'explosion d'états et nécessitent des modèles avec un nombre restreint d'états.

Après la modélisation des systèmes TR^{2}E dynamiquement reconfigurables et la vérification de certaines propriétés non-fonctionnelles au niveau modèle de conception, une phase de génération de code est demandée afin de concevoir facilement ces systèmes. Dans la section suivante, nous décrivons quelques intergiciels dédiés aux systèmes embarqués.

TABLE 2.3 – Tableau récapitulatif des approches existantes/propriétés

Frameworks	Respect des échéances	Cons. CPU	Cons. mémoire	Cons. bande passante	Absence d'inter-blocage	Absence de famine
VERTAF [22]	✓	✘	✘	✘	✓	✓
Cheddar [56]	✓	✓	✘	✘	✘	✘
Dream [40]	✓	✘	✓	✘	✓	✘
Tasm Toolset [49]	✘	✓	✓	✓	✓	✘
ModSyn [10]	✘	✘	✘	✘	✓	✘

2.5 Intergiciels dédiés aux systèmes embarqués

Face à la complexité sans cesse grandissante des systèmes embarqués, des intergiciels offrant un ensemble de fonctionnalités pour faciliter le développement de ces systèmes sont nécessaires. Dans ce cadre, plusieurs intergiciels ont été proposés pour couvrir les besoins des systèmes TR^{2}E. Dans cette section, nous présentons ceux qui sont les plus proches de notre approche.

2.5.1 DynamicTAO

DynamicTAO [26] est une extension de l'intergiciel TAO [54] qui est un ORB reconfigurable, portable, flexible, extensible et basé sur les patrons de modélisation orientés objet. En effet, TAO utilise la stratégie de modélisation par des patrons afin de séparer les différents aspects du moteur interne de l'ORB. Il est dédié aux applications temps réel statiques.

Dynamic TAO a été introduit pour supporter des applications adaptatives qui s'exécutent sur des environnements dynamiques. Il repose sur un ORB conforme à CORBA. Il implante les mécanismes de reconfiguration dynamique pour des systèmes répartis et performants. Il implante aussi les mécanismes de concurrence, de sécurité et de supervision au cours de l'exécution.

DynamicTAO est un ORB réflexif. En effet, il permet la supervision et la reconfiguration de son moteur interne. Il assure la réflexivité par l'exportation d'une interface permettant de transférer des composants à travers des systèmes répartis, charger/décharger des modules dans l'ORB au cours de l'exécution, et superviser et modifier l'état de configuration de l'ORB. Afin d'assurer facilement la reconfiguration dynamique, les composants du système sont regroupés dans des bibliothèques chargées et utilisées au cours de l'exécution. Dans dynamicTAO, la réflexivité est

assurée à travers un ensemble d'entités appelées composants configurateurs. Pour chaque processus en cours d'exécution sur dynamicTAO, une instance du composant configurateur est créée.

DynamicTAO préserve la cohérence du système lors du déchargement des composants. Il vérifie que toutes les invocations du composant sont traitées avant le déchargement et il assure la préservation de l'état du composant par l'envoi d'une partie de l'état interne du composant déchargé vers un nouveau composant. DynamicTAO assure aussi la sécurité lors des reconfigurations dynamiques. Pour cela, il offre une architecture de sécurité flexible par l'injection d'agents assurant la supervision et la reconfiguration.

Discussion :

DynamicTAO est un intergiciel supportant les systèmes embarqués temps réel dynamiquement reconfigurables. Il assure la reconfiguration dynamique ainsi que la supervision et la cohérence de ces systèmes. Il peut alors facilement s'adapter à l'environnement d'exécution du système. Cependant, DynamicTAO ne supporte que des reconfigurations architecturales. De plus, DynamicTAO n'est pas bien adapté pour des systèmes temps réel durs (critiques). Il n'assure pas la vérification des contraintes temporelles pendant et après des reconfigurations.

2.5.2 CIAO

CIAO (Component Integrated ACE ORB) [58] est un intergiciel supportant des systèmes TR^{2}E basés sur les composants. Il propose une implantation libre du modèle de composant LwCCM [44] et de la spécification RT-CORBA (Real Time CORBA) [41]. Il offre des mécanismes permettant la spécification, l'implantation, l'assemblage et le déploiement des composants. De plus, CIAO supporte les contraintes additionnelles sur les temps d'initialisation du système et les fonctions disponibles (comme par exemple la liaison et le chargement dynamique).

En effet, un framework de déploiement et de configuration a été intégré dans l'intergiciel CIAO afin d'assurer la gestion du déploiement et de la configuration des composants de QoS ainsi que des services de l'intergiciel.

Par ailleurs, CIAO assure une robuste QoS grâce à la séparation de l'implantation des composants du déploiement et de la configuration. En effet, il a recours aux techniques à base d'aspects afin d'assurer la séparation et la composition des aspects temps réel et des préoccupations propres à la configuration.

Afin d'adresser les défis de performance et de faisabilité de déploiement et de

reconfiguration (D&C) du système, CIAO étend la définition du conteneur d'un composant et les capacités de représentation et de manipulation des méta-données du composant.

Discussion :

CIAO est un intergiciel adapté aux systèmes embarqués temps réel et flexibles. Il permet de simplifier et d'automatiser le (re)déploiement et la (re)configuration. Cependant, ni la nature ni l'implantation des mécanismes de reconfiguration ne sont spécifiées. Par ailleurs, CIAO ne prend pas en charge l'ordonnancement des tâches apériodiques. Il supporte seulement le modèle de composants LwCCM.

2.5.3 SwapCIAO

SwapCIAO [3] est une extension de l'intergiciel CIAO visant à supporter la reconfiguration dynamique des systèmes TR^{2}E. Il permet la mise à jour dynamique des implantations de composants grâce aux extensions fournies par le modèle de composant LwCCM.

La reconfiguration dynamique dans SwapCIAO consiste à faire une mise à jour des implantations de composants. Elle se fait selon les étapes suivantes : la désactivation de l'implantation du composant, le retrait de l'implantation du composant de la plate-forme d'exécution et la modification de l'implantation du composant. En effet, SwapCIAO assure :

- des mises à jour continues et consistantes en s'assurant que le composant à reconfigurer a terminé les invocations en cours et que les nouvelles invocations sont bloquées jusqu'à la fin de la mise à jour du composant.
- la transparence des mises à jour en garantissant la redirection des nouvelles invocations vers la nouvelle implantation du composant.
- la reconnexion des composants par la restauration des connexions après la mise à jour de l'implantation du composant. Ces connexions sont déjà stockées dans des descripteurs XML lors du déploiement de l'application.

Discussion :

SwapCIAO est un intergiciel flexible, performant et adapté aux systèmes TR^{2}E dynamiquement reconfigurables. Il offre aussi des garanties de QoS. Par contre, SwapCIAO est très difficile à maintenir. Pour cette raison, il n'y a pas jusqu'à maintenant d'implantation pour l'intergiciel SwapCIAO.

2.5.4 FLARe

FLARe (Fault-tolerant Load-aware and Adaptive middlewaRe) [4] est un intergiciel qui étend TAO [54]. Il supporte les applications réparties temps réel molles. Il permet aussi la gestion de la tolérance aux pannes et le recouvrement des exigences des systèmes temps réel mous. FLARe assure la reconfiguration dynamique selon la disponibilité des ressources. Il offre des mécanismes de reconfiguration qui consistent en la redirection des clients en cas de panne (surcharge de processeur).

Afin de fournir des mécanismes de tolérance aux pannes, FLARe utilise la réplication passive qui consiste à traiter toutes les requêtes par un seul serveur appelé "primaire". Lorsque ce dernier tombe en panne, il est remplacé par un autre serveur.

Discussion :

FLARe implante des mécanismes assurant la gestion des pannes et des surcharges de façon transparente aux clients. Cependant, il ne supporte pas les systèmes temps réel durs. De plus, il ne gère que les tâches périodiques et sa maintenance est très complexe.

2.5.5 PolyORB_HI

PolyORB_HI [63] est un intergiciel inspiré de l'architecture de PolyORB [59]. Il utilise l'architecture schizophrène et ses services canoniques afin d'assurer la communication entre plusieurs plates-formes hétérogènes. PolyORB_HI est conforme au Profil Ravenscar [6, 36]. Il respecte les restrictions proposées par ce profil qui restreignent l'usage de certaines constructions des langages de programmation (ADA, C, etc.) afin d'assurer l'ordonnancement du système ainsi que l'absence d'interblocage.

PolyORB_HI est composé d'un noyau représentant l'intergiciel minimal et plusieurs services générés automatiquement. L'intergiciel minimal offre les services communs pour toutes les applications tandis que les services générés automatiquement à partir d'une spécification AADL sont adaptables aux exigences de l'application cible. Grâce à sa faible empreinte mémoire, PolyORB_HI représente un intergiciel idéal pour les systèmes embarqués.

Discussion :

PolyORB_HI présente l'avantage d'être portable, sûr et déterministe. Cependant, dans cet intergiciel, les composants statiquement configurés (intergiciel minimal) doivent être compilés avec ceux qui sont produits automatiquement. Par ailleurs, il n'implante pas les mécanismes assurant la reconfiguration dynamique des

	Intergiciels				
critères	DynamicTAO	CIAO	PolyORB_HI	FLARe	SwapCIAO
Granularité	Objet (RT-CORBA)	Composant (LwCCM)	Composant	Objet (RT-CORBA)	Objet (LwCCM)
Hard real-time	✘	✓	✓	✘	N/A
Taille mémoire	1,5 MB	5 MB	70 KB	10 MB	N/A
Supervision	✓	N/A	✘	✓	✓
Reconf. dynamique	✓	✓	✘	✓	✓
Portabilité	✓ (C++)	✓ (C++)	✓ (Java)	✓ (C++)	✓ (C++)
Coherence	✓ (difficult)	N/A	✘	✘	✓

TABLE 2.4 – Intergiciels pour des systèmes embarqués

systèmes TR^{2}E.

2.5.6 Synthèse

Dans le tableau 2.4, nous résumons les propriétés les plus importantes des intergiciels précédemment présentés. Nous visons à proposer un intergiciel pour supporter la reconfiguration dynamique ainsi que la supervision et la cohérence comme les deux intergiciels DynamicTAO et SwapCIAO. Cependant, ces deux derniers supportent seulement les systèmes temps réel mous basés objet. Notre objectif est de concevoir un intergiciel supportant des systèmes embarqués temps réel basés composant comme c'est le cas pour les deux intergiciels PolyORB_HI et CIAO. Mais ces derniers n'assurent ni la supervision ni la cohérence. Contrairement à PolyORB_HI, CIAO assure la reconfiguration dynamique. Mais il a une grande empreinte mémoire ($\simeq$ 5 MB). PolyORB_HI a une faible empreinte mémoire ($\simeq$ 70 KB). Compte tenu de l'existant présenté ci-dessus, nous visons à développer un intergiciel traitant des reconfigurations architecturales et comportementales avec une faible empreinte mémoire.

Dans la section suivante, nous présentons quelques méthodologies et processus de développement assurant le développement des systèmes embarqués.

2.6 Processus et frameworks de développement

Pour faire face à la complexité croissante de la conception et du développement des systèmes embarqués, de nouveaux processus de développement ont été proposés. Ces processus offrent des méthodologies permettant de spécifier ces systèmes en prenant en considération leurs contraintes non-fonctionnelles. Ces méthodologies permettent d'une part de diminuer la complexité de la mise en oeuvre de ces systèmes et d'autre part de réduire le coût de développement. Dans la suite, nous décrivons des processus de développement et frameworks proposés pour la spécification et le développement des systèmes embarqués.

2.6.1 ModES

ModES (Model-driven design approach) [9] est une approche basée sur l'ingénierie dirigée par les modèles (IDM). Elle présente une méthodologie et offre un ensemble d'outils de conception, d'estimation et de génération de code permettant de concevoir des systèmes embarqués. Cette méthodologie consiste, en une première phase, à transformer des modèles d'application et des modèles de plate-forme spécifiés par des langages de modélisation (UML) en des modèles conformes respectivement au méta-modèle d'application interne IAMM (*Internal Application Meta-Model*) et au méta-modèle de plate-forme interne IPMM (*Internal Platform Meta-Model*). Le mapping entre ces derniers modèles (l'allocation de l'application sur la plate-forme), qui est conforme au méta-modèle de mapping MMM (Mapping Meta-Model), est assuré dans une deuxième phase. Dans une troisième phase, les modèles sont transformés en des modèles d'implantation conformes à un méta-modèle d'implantation IMM (Implementation Meta-Model). Finalement, une partie du code source, des scripts de synthèse matérielle, et des scripts de déploiement du système sont générés à partir des modèles d'implantation. Des outils d'analyse sont aussi utilisés afin de fournir des estimations sur les propriétés physiques du système (e.g. cycles d'exécution, consommation d'énergie, empreinte mémoire, etc.) au niveau modèle de mapping.

ModES définit donc quatre nouveaux méta-modèles : un méta-modèle IAMM pour la spécification haut niveau de l'application, un méta-modèle IPMM pour la spécification de la plate-forme d'exécution, un méta-modèle de mapping présentant les règles d'allocation de l'application sur la plate-forme, et un méta-modèle d'implantation permettant la génération de code. L'évaluation des implantations pos-

sibles au cours du processus de conception en se basant sur des estimations précises de chaque séquence de transformation est assurée au niveau modèle de mapping. La transformation d'un modèle vers un autre est définie en utilisant le langage de transformation de modèles QVT [47] qui est un standard de l'OMG.

Discussion :

ModES est une approche basée sur l'ingénierie dirigée par les modèles pour des systèmes embarqués. Afin d'assurer la modélisation des différents concepts de ces systèmes, une infrastructure de méta-modélisation est proposée.

Cependant, la plupart des nouveaux concepts définis dans les méta-modèles proposés sont déjà définis dans le profil UML MARTE qui est un standard assurant la modélisation et l'analyse des systèmes embarqués temps réel [45]. Le profil MARTE peut donc permettre la modélisation du modèle interne de l'application, le modèle interne de la plate-forme et le modèle de mapping. Il est même plus riche que ces méta-modèles proposés. De plus, cette approche ne définit pas d'intergiciel facilitant la génération d'une grande partie de code à partir du modèle d'implantation en offrant un ensemble de routines. En outre, ModES ne supporte pas les systèmes dynamiquement reconfigurables. Il définit un processus de développement permettant de concevoir des systèmes embarqués statiques.

2.6.2 Framework dirigé par les modèles [8]

Un framework basé sur les architectures dirigées par les modèles [8] a été proposé afin de concevoir des systèmes temps réel. Ce framework s'appuie sur une méthodologie qui facilite la modélisation et l'implantation de ces systèmes sur différentes plate-formes d'exécution. Dans cette approche, un modèle indépendant de la plate-forme (PIM) est modélisé en utilisant le langage UML et le sous-profil HLAM (High-Level Application Modeling) du profil MARTE. Le modèle PIM est transformé en un modèle spécifique à la plate-forme (PSM) par des règles de transformation. Pour chaque plate-forme cible, pour obtenir le modèle PSM à partir du modèle PIM, il faut avoir des règles de transformation spécifiques. Afin de spécifier des règles de transformation génériques permettant d'obtenir différents modèles PSM à partir du modèle PIM, les modèles des différentes plates-formes doivent être conformes au même méta-modèle. Pour cela, le modèle PIM est enrichi par des modèles en utilisant le sous-profil SRM (Software Resource Modeling) du profil MARTE permettant de spécifier les ressources et les services des systèmes d'exploitation et le profil DPD (Detailed Platform Description). A partir du modèle PIM et en utilisant les règles

de transformation génériques, le modèle PSM est obtenu. A partir du modèle PSM et en utilisant des générateurs de code, le code applicatif correspondant est généré.

Discussion :

Cette approche orientée architecture et basée sur les modèles a été proposée afin de générer le code applicatif des systèmes embarqués temps réel. L'apport de cette approche est qu'elle offre des règles de transformation génériques permettant d'obtenir un modèle PSM à partir d'un modèle PIM. Ces règles de transformation permettent d'obtenir différents PSM à partir d'un PIM spécifié en utilisant les deux sous-profils HLAM et SRM du profil MARTE et le profil DPD. Cependant, cette approche ne prend pas en considération les reconfigurations dynamiques dans les systèmes embarqués temps réel. En outre, la modélisation du PIM repose seulement sur les deux sous-profils HLAM et SRM du profil MARTE.

2.6.3 CoSMIC

CoSMIC [11, 12] est une suite d'outils proposés pour la composition et le déploiement des applications TR^{2}E et respectant le standard D&C [43]. Cette suite d'outils permet tout d'abord la modélisation et l'analyse des fonctionnalités des applications TR^{2}E et les exigences de qualité de service (QoS). Afin d'offrir et appliquer la qualité de service, elle permet de générer les méta-données de déploiement spécifiques au CCM [44] pour les intergiciels CIAO et QuO. Un processus de développement des applications TR^{2}E basé sur la suite d'outils CoSMIC est proposé afin de modéliser les exigences et les politiques d'adaptation nécessaires pour la gestion de la QoS de ces applications. Ce processus permet l'assemblage des composants logiciels de l'intergiciel en assurant une compatibilité entre leurs différents paramètres de qualité de service. Après la modélisation de l'assemblage et du déploiement des composants en utilisant le langage CADML (Component Assembly and Deployment Modeling Language), des plans de déploiement sont générés. Ces plans de déploiement décrivent les différentes possibilités de déploiement pour une telle application. CoSMIC utilise aussi un langage spécifique OCML (Options Configuration Modeling Language) pour la configuration des paramètres et des contraintes. Il génére les méta-données de configuration d'intergiciels des applications TR^{2}E.

Discussion :

CoSMIC est une suite d'outils permettant l'automatisation de la configuration et du déploiement des applications TR^{2}E. L'automatisation de la configuration et du déploiement permet de réduire considérablement les risques d'erreurs.

Cependant, CoSMIC utilise différents langages pour la configuration et le déploiement des composants des applications TR²E ainsi que pour la configuration de l'application et du middleware (CADML, OCML). Le concepteur doit donc avoir des compétences sur ces langages afin de modéliser correctement son application. De plus, CoSMIC ne prend pas en considération les reconfigurations dynamiques dans les applications TR²E. En outre, CoSMIC repose sur l'intergiciel CIAO qui ne supporte pas les spécificités des systèmes critiques. En effet, cet intergiciel se base sur l'intergiciel TAO qui assure une allocation dynamique de la mémoire. De plus, CIAO ne supporte que le modèle de composant CCM [44].

2.6.4 Ocarina

Ocarina [23, 37] est un ensemble d'outils permettant de concevoir des systèmes embarqués temps réel répartis. Il offre un framework assurant le développement, la configuration et le déploiement des systèmes TR²E.

Ce framework consiste, tout d'abord, à modéliser des systèmes TR²E statiques en utilisant le langage de description d'architecture AADL. A partir des modèles AADL et comme le montre la Figure 2.2, Ocarina assure les analyses lexicale, syntaxique et sémantique afin de garantir la confirmité des modèles à la grammaire AADL [52]. Si les analyses sont faites avec succès, une instanciation du système est nécessaire. Grâce à son organisation en bibliothèques logicielles, Ocarina permet aussi de faire manipuler des modèles AADL par d'autres outils comme par exemple l'outil d'analyse statique d'ordonnancement Cheddar [56].

A partir des modèles AADL, les générateurs de code d'Ocarina produisent une grande partie du code applicatif ainsi qu'une couche de l'intergiciel consacrée à des besoins spécifiques à l'application. Cette dernière rassemble les services canoniques fortement personnalisés pour l'application. Ocarina permet de générer des constructions des langages de programmation Ada et C qui sont conformes aux restrictions du profil Ravenscar [6] à partir des entités AADL. Pour cela, deux versions de l'intergiciel PolyORB_HI ont été proposées, une en Ada et l'autre en C. L'intergiciel PolyORB_HI présente la couche minimale qui implante les services canoniques faiblement personnalisables par l'application. Une autre couche de l'intergiciel présentant les services canoniques fortement personnalisables est générée avec le code applicatif de l'application.

Ocarina assure automatiquement le déploiement et la configuration de l'application à l'aide des informations extraites à partir des modèles AADL.

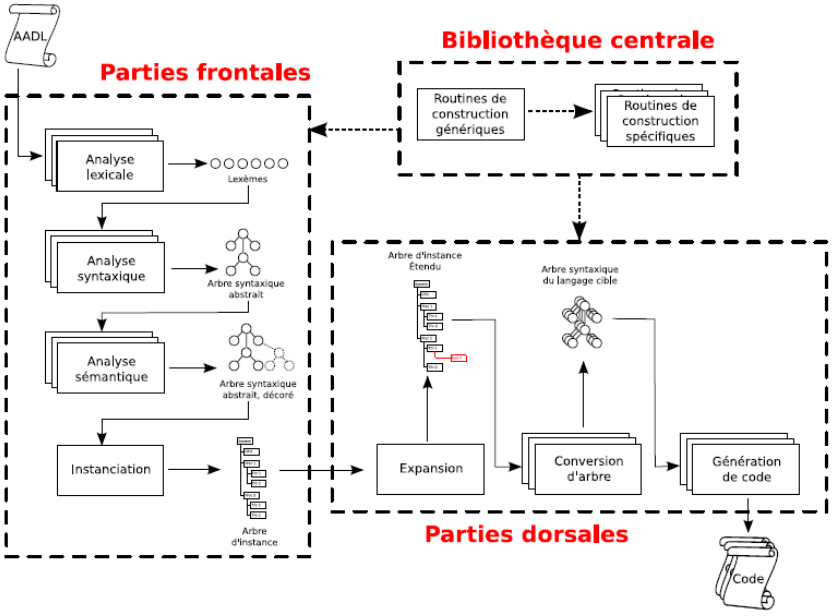

FIGURE 2.2 – Architecture globale d'Ocarina [62]

Discussion :

Ocarina est un framework permettant la manipulation des modèles AADL afin d'effectuer des analyses et vérifications [61] au niveau modèle. Il assure aussi la génération automatique de code à partir des modèles AADL en générant des constructions des langages de programmation Ada et C. Pour cela, un nouveau intergiciel PolyORB_HI a été proposé en deux versions, une en C et l'autre en Ada.

Cependant, le framework Ocarina traite seulement les systèmes TR^{2}E statiques et ne prend pas en considération les systèmes dynamiquement reconfigurables. De plus, l'utilisation du langage AADL pour la modélisation des systèmes TR^{2}E fournit une modélisation à un niveau concret (threads, processeurs, etc) et pas à un haut niveau d'abstraction.

2.6.5 TimeAdapt

TimeAdapt [15] est un framework permettant d'assurer des reconfigurations dynamiques pour des systèmes embarqués. Il repose sur trois phases. La première phase consiste à fournir des moyens permettant la spécification de l'ensemble des actions de reconfiguration à effectuer. La deuxième phase permet l'estimation du temps d'exécution de ces actions. Et la troisième phase assure l'exécution des actions de reconfiguration dynamique. En effet, ce framework possède un gestionnaire de reconfiguration. Ce dernier est responsable de la réception des actions de reconfigura-

tion spécifiées en utilisant un langage de reconfiguration permettant d'exprimer les contraintes de reconfiguration. Chaque reconfiguration doit respecter ses contraintes temporelles (limites de temps) afin d'être exécutée. En effet, le temps d'exécution de chaque reconfiguration dépend des conditions environnementales et structurelles de l'application.

Pour cela, un test d'admission se charge de calculer la probabilité pour qu'une reconfiguration donnée respecte ses contraintes temporelles. Si la valeur de la probabilité calculée dépasse un seuil donné, la reconfiguration sera exécutée après la génération des constructions de reconfiguration en langage d'exécution. Elle sera considérée comme une tâche temps réel de haute priorité et les actions de reconfiguration seront donc exécutées. Sinon (la valeur de la probabilité calculée ne dépasse pas un seuil donné), les contraintes temporelles de la reconfiguration doivent être modifiées afin d'exécuter la reconfiguration plus tard.

Discussion :

TimeAdapt est un framework assurant les reconfigurations dynamiques pour des systèmes embarqués temps réel, ce qui permet l'adaptation de ces systèmes à leur environnement et l'amélioration de leurs temps de réponse. Il permet de générer le code source des reconfigurations spécifiées vers différentes plates-formes d'exécution en utilisant différents modèles (templates) de génération (comme par exemple l'outil de translation direct de syntaxe StringTemplate [1] pour générer du code Java). De plus, ce framework supporte le modèle de composant UML2 [46].

Cependant, ce framework définit des limites de temps pour chaque reconfiguration. Il calcule le probabilité qu'une reconfiguration respecte ses limites temporelles. Si la probabilité montre la possibilité qu'une reconfiguration dépasse son délai estimé, la reconfiguration ne sera pas exécutée. Le système peut donc avoir besoin d'une reconfiguration sans avoir la possibilité d'être reconfiguré.

En outre, le langage utilisé pour spécifier les actions de reconfiguration ainsi que les contraintes de reconfiguration a des inconvénients. D'une part, les utilisateurs de TimeAdapt doivent donc bien maîtriser ce langage afin d'utiliser ce framework. D'autre part, ce langage ne permet pas de spécifier les parties logicielles et matérielles des systèmes embarqués ainsi que l'allocation de la partie logicielle sur la partie matérielle de chaque système.

2.6.6 COMDES

COMDES (*COMponent-based design of software for Distributed Embedded Systems*) [19] est un framework dédié à la spécification et la configuration des systèmes embarqués contraints en ressources. Ce framework a été conçu afin de résoudre les problèmes de conception des systèmes embarqués temps réel en tenant compte de leurs ressources limitées. COMDES définit un processus de développement commençant par la modélisation des systèmes embarqués jusqu'à la production du code de l'application. Un système est modélisé dans son domaine d'application à un haut niveau d'abstraction. Puis le modèle conçu est transformé en un modèle COMDES qui est généralement enrichi par des informations qui guident à la génération du code. Finalement, le code généré est déployé et testé. Dans ce framework, l'application est développée et configurée à partir de composants préfabriqués stockés dans un entrepôt de composants.

Ce framework supporte un processus de configuration et un processus de reconfiguration. Le processus de configuration consiste à sélectionner les composants à partir d'un entrepôt de composants préfabriqués et à assembler ces composants afin de construire une configuration de l'application. Le processus de reconfiguration permet l'addition, la suppression et la modification des composants.

Discussion :

COMDES est un framework de modélisation des systèmes embarqués répartis contraints en ressources à un haut niveau d'abstraction. Les modèles de haut niveau sont transformés en un modèle COMDES qui permet la production du code applicatif. Cependant, en utilisant le framework COMDES, le développeur possède un nombre limité de composants préfabriqués qui sont stockés dans un entrepôt. Le développeur ne peut donc pas ajouter un nouveau composant qui n'existe pas dans l'entrepôt. De plus, COMDES ne prend pas en considération la partie matérielle des systèmes embarqués. Il ne couvre ni la modélisation ni la vérification des ressources matérielles. Il ne vérifie pas si les ressources matérielles supportent les composants logiciels utilisés. En outre et bien que COMDES assure la génération de code, il ne dispose pas d'un intergiciel supportant le code généré.

2.6.7 Synthèse

Plusieurs frameworks et processus ont été proposés afin de concevoir des systèmes embarqués temps réel. Les approches proposées présentent des méthodologies allant de la phase de modélisation jusqu'à la phase d'implantation.

Comme le montre le tableau 2.5, certaines approches [9, 12, 23, 37] supportent des systèmes répartis mais qui ne sont pas reconfigurables. Parmi ces approches, [9] assure la modélisation des systèmes embarqués temps réel à plusieurs niveaux d'abstraction et la génération de code mais elle n'offre pas un intergiciel facilitant la génération. [12, 23, 37] permettent de générer une grande partie du code en utilisant des intergiciels mais elles n'assurent pas la modélisation à plusieurs niveaux d'abstraction.

L'approche proposée dans [8] présente un processus de développement pour des systèmes embarqués mais qui ne sont ni répartis ni reconfigurables. Elle génère une grande partie du code en se basant sur un intergiciel mais elle n'assure pas la modélisation à plusieurs niveaux d'abstraction.

Des processus de développement [15, 19] des systèmes répartis reconfigurables ont été proposés mais ces processus ne proposent pas d'intergiciel facilitant la génération du code.

TABLE 2.5 – Tableau comparatif des Frameworks/processus et leurs domaines d'utilisation

Approches	Systèmes répartis	Systèmes reconfigurables	Modélisation à plusieurs niveaux d'abstraction	Intergiciel	Génération de code
Framework [8]	✘	✘	✓	✘	✓
COMDES [19]	✓	✓	✓	✘	✓
TimeAdapt [15]	✓	✓	✘	✘	✓
OCARINA [23, 37]	✓	✘	✘	✓	✓
ModES [9]	✓	✘	✓	✘	✓
CoSMIC [12]	✓	✘	✘	✓	✘

Pour cela, nous proposons une approche IDM permettant la modélisation des systèmes TR^{2}E à plusieurs niveaux d'abstraction en passant d'un modèle abstrait vers un modèle plus détaillé jusqu'à la génération de code. Afin de générer une grande partie du code, nous avons développé un intergiciel supportant les systèmes TR^{2}E dynamiquement reconfigurables.

2.7 Conclusion

Dans ce chapitre, nous avons présenté un état de l'art sur le développement des systèmes embarqués répartis couvrant les étapes de modélisation, de vérification, de génération et d'implantation. Pour ce faire, nous avons d'abord commencé par la des-

cription des outils de modélisation des systèmes embarqués. Nous avons également étudié les différents travaux permettant d'assurer la vérification de ces systèmes au niveau modèle de conception. Ensuite, nous avons étudié les différents intergiciels dédiés aux systèmes embarqués. Nous avons présenté les principaux processus et frameworks de développement pour ces systèmes.

Dans le chapitre suivant, afin de pallier les faiblesses des travaux existants, nous proposons un processus de développement automatisé des systèmes TR^2E dynamiquement reconfigurables.

Chapitre 3

Modélisation des systèmes TR^{2}E dynamiquement reconfigurables

3.1 Introduction

Dans l'état de l'art présenté dans le chapitre précèdent, nous avons analysé l'existant en matière de processus et frameworks de développement des systèmes embarqués. Nous avons aussi décrit les outils liés à la modélisation des systèmes embarqués ainsi qu'à la vérification de certaines propriétés non-fonctionnelles au niveau modèle de conception. L'accent a été mis aussi sur les intergiciels dédiés aux systèmes embarqués.

Partant de l'étude effectuée, nous avons souligné des manques dans les approches existantes. Ceci peut se résumer par :

- Le manque de méthodologies pour les spécifier et les implanter,
- Le manque de concepts permettant la modélisation de ces systèmes et des reconfigurations dynamiques,
- Le manque d'outils de vérification au niveau modèle de conception,
- Le manque d'intergiciels dédiés aux systèmes temps réels dynamiquement reconfigurables, etc.

A l'issue de ce constat, nous proposons dans ce chapitre une solution aux problématiques citées dans l'introduction générale (Chapitre 1) qui vise à surmonter les limites des solutions étudiées dans l'état de l'art (Chapitre 2). Il s'agit d'introduire une approche à base de modèles pour la spécification et la gestion de la reconfiguration dynamique des systèmes embarqués temps-réel.

Nous définissons, dans la section 3.2, le processus de développement proposé

couvrant les phases de modélisation, d'allocation, de vérification et de génération du code pour une plate-forme d'exécution. La section 3.3 présente l'infrastructure de modélisation de notre approche. L'accent est mis sur le profil MARTE (section 3.4) et sur le langage de modélisation choisi pour supporter la phase de modélisation. Enfin, la section 3.6 conclut ce chapitre.

3.2 Processus de développement des systèmes TR²E dynamiquement reconfigurables

Nous décrivons dans ce qui suit le processus de développement à base de modèles [30, 34] proposé pour des systèmes TR²E dynamiquement reconfigurables comme le montre la Figure 3.1. Ce processus est représenté par un diagramme d'activité d'UML. Il est composé principalement de quatre phases :

1. Phase de modélisation : elle consiste à spécifier le système TR²E dynamiquement reconfigurable à un haut niveau d'abstraction en passant par les quatre étapes suivantes :
   - Spécification de la reconfiguration,
   - Spécification de la partie logicielle : spécification de l'architecture logicielle en termes de composants,
   - Spécification de la partie matérielle : spécification de l'architecture matérielle en termes de composants matériels tels que la mémoire, le bus, le processeur, etc.
   - Spécification de l'allocation logiciel/matériel : spécification de l'allocation de chaque type de configuration sur une architecture matérielle.
2. Phase de vérification : elle décrit la vérification de certaines propriétés non-fonctionnelles au niveau modèle de conception.
3. Phase de génération I : elle consiste à décrire la génération des différentes configurations (modes) du système à partir des modèles de haut niveau pour configurer la plate-forme d'exécution, et le modèle d'implantation facilitant la génération du code pour une plate-forme d'exécution cible.
4. Phase de génération II : elle décrit la génération du code pour la plate-forme d'exécution ainsi configurée via les modèles générés préalablement.

Dans le reste de ce chapitre, nous allons détailler le langage de modélisation et son fondement. Dans le chapitre 4), nous passons à présenter les outils de vérification

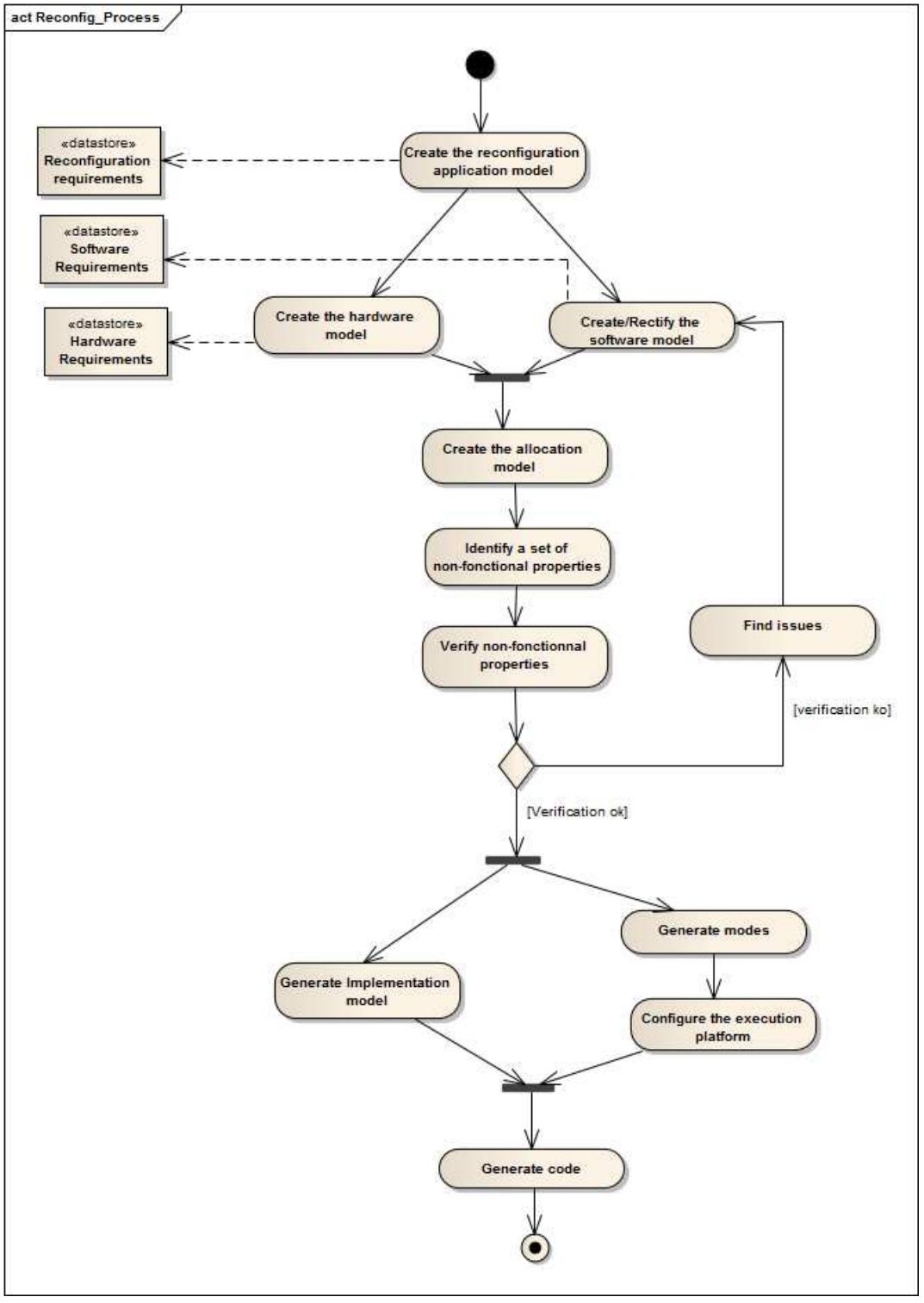

FIGURE 3.1 – Processus de développement de systèmes TR²E reconfigurables

et leurs principes pour développer le framework de vérification. Par la suite, nous abordons la plate-forme d'exécution dédiée à ces systèmes dans le chapitre 5.

3.3 Architecture de l'infrastructure de modélisation

L'approche proposée utilise des techniques de modélisation, de méta-modélisation et de transformation de modèles pour spécifier les systèmes TR²E dynamiquement reconfigurables. Pour ce faire, nous définissons de nouveaux concepts, des nouveaux méta-modèles tout en favorisant la réutilisation des méta-modèles existants (voir Figure 3.2) :

1. Méta-modèles existants :
   - UML : nous utilisons les machines à états d'UML pour spécifier les modes et les politiques de reconfiguration,
   - MARTE : nous adoptons les concepts de MARTE pour identifier les architectures logicielles et matérielles ainsi que les contraintes non-fonctionnelles.
2. Notre proposition :
   - Un méta-modèle pour la spécification des systèmes TR²E et en particulier la reconfiguration (voir la section 3.5). Ce méta-modèle est combiné au profil MARTE pour spécifier un système TR²E dynamiquement reconfigurable à un haut niveau d'abstraction.
   - Un méta-modèle pour l'implantation (voir la section 5.5).

3.4 Profil MARTE

La Figure 3.3 montre les différents paquetages ou sous profils du profil MARTE [45]. A travers ces paquetages, MARTE permet de spécifier des systèmes embarqués temps réel sur plusieurs niveaux d'abstraction. Le profil MARTE est formé principalement de trois paquetages :

1. "*MARTE foundation*" : il regroupe les concepts fondamentaux pour la modélisation et l'analyse des systèmes embarqués temps réel. Il est composé des sous paquetages suivants :
   - *CoreElements* : il contient les éléments de base. C'est le noyau du profil MARTE,
   - *NFP* (*Non-Functional Properties*) : il permet de spécifier les propriétés et les contraintes non-fonctionnelles,
   - *Time* : il contient les différents concepts nécessaires pour définir le temps,
   - *GRM* (*Generic Resource Modeling*) : il fournit des concepts de base pour modéliser une plate-forme générale (logicielle et matérielle) selon différents

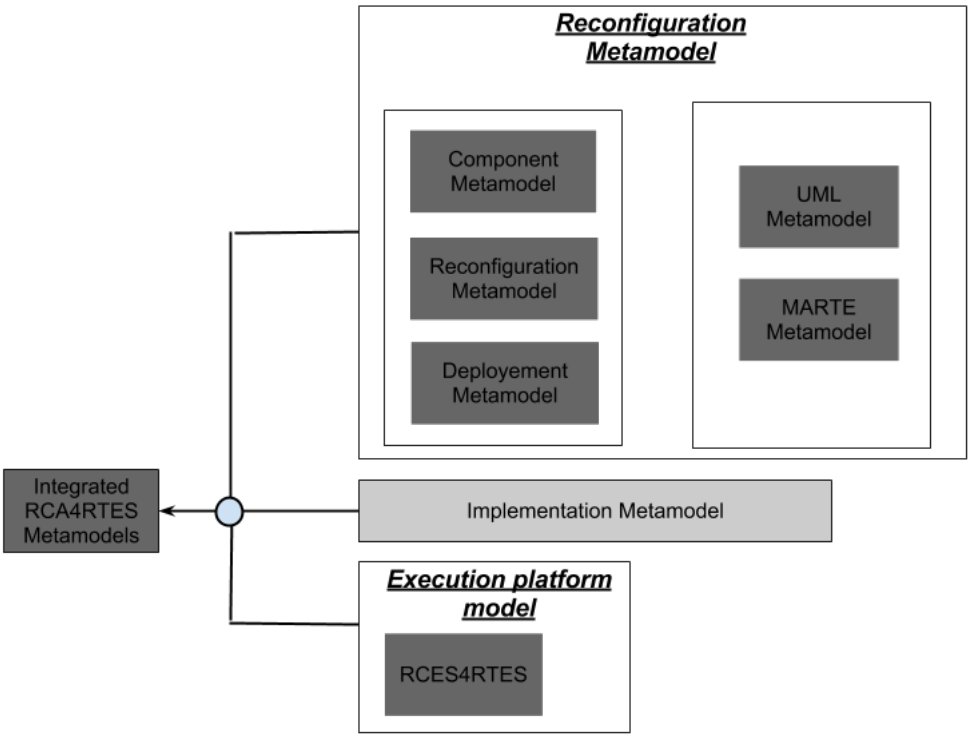

FIGURE 3.2 – Architecture de méta-modélisation

niveaux d'abstraction,

- *Alloc* : il permet d'allouer l'architecture logicielle sur l'architecture matérielle du système, ainsi que la projection des entités fonctionnelles (composants) sur des ressources physiques.

2. "*MARTE design model*" : il se base sur les concepts fondamentaux du profil MARTE pour la modélisation des systèmes embarqués temps réel. Il est composé des sous-paquetages suivants :
   - *GCM* (*Generic Component Model*) : il supporte l'ingénierie à base de composants. Contrairement à UML2 qui définit seulement les composants et les connexions entre eux, *GCM* identifie aussi les flux de contrôle et de données,
   - *HLAM* (*High-Level Application Modeling*) : il fournit des concepts de modélisation de haut niveau pour assurer la modélisation des caractéristiques des systèmes embarqués temps réel,
   - *SRM* (*Software Resource Modeling*) : il permet de décrire des APIs (Application Programming Interfaces) pour des supports d'exécution logiciel des systèmes embarqués temps réel,

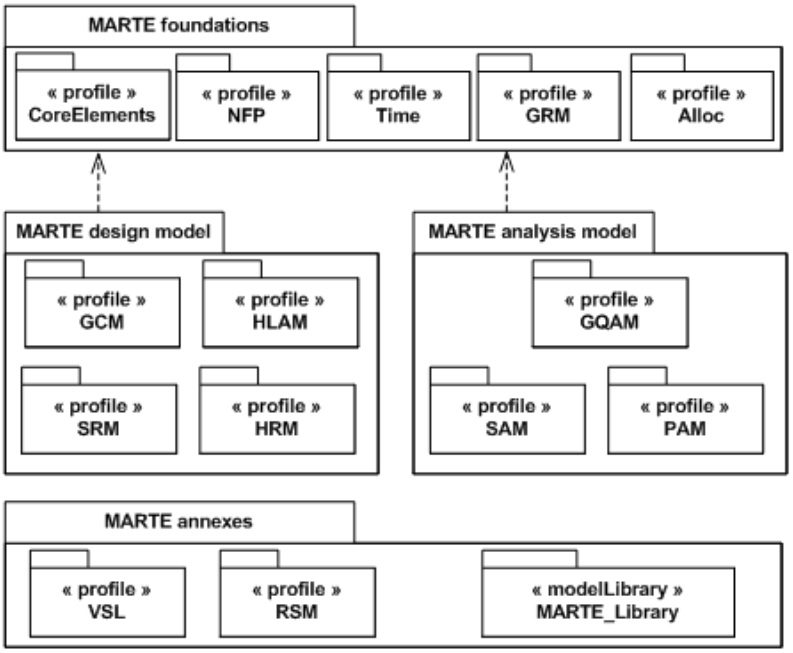

FIGURE 3.3 – Architecture du profil MARTE [45]

- *HRM* (*Hardware Resource Modeling*) : il permet de spécifier la partie matérielle du système.

3. "*MARTE analysis model*" : il spécialise les concepts fondamentaux du profil MARTE pour l'analyse des systèmes embarqués temps réel. Il est composé de trois sous paquetages :
   - *GQAM* (*Generic Quantitative Analysis Model*) : il regroupe des concepts définissant l'analyse quantitative générique,
   - *SAM* (*Schedulability Analysis Model*) : il permet de spécifier l'analyse d'ordonnancement,
   - *PAM* (*Performance Analysis Model*) : il contient des concepts permettant l'analyse de la performance.

D'autre part, le profil MARTE propose un paquetage "*MARTE Annexes*", composé d'un ensemble de sous paquetages utiles, pour la modélisation et l'analyse des systèmes embarqués temps réel :

- *VSL* (*Value Specification Language*) : il s'agit d'un langage pour la spécification des contraintes non-fonctionnelles. C'est une extension du langage déclaratif OCL pour supporter la spécification des contraintes comportementales des systèmes. Il présente des expressions mathématiques (arithmétique, logique, etc.) et des expressions du temps complexes (délais, période, conditions de

déclenchement, etc.),
- *RSM* (Repetitive Structure Modeling) : il propose des constructions assurant la modélisation haut niveau des systèmes embarqués de calcul intensif tels que les systèmes de traitement de signal ou d'image,
- *MARTE_Library* : il définit de nouveaux types utiles pour spécifier les propriétés non-fonctionnelles des systèmes embarqués temps réel.

3.5 Langage de modélisation des systèmes TR²E dynamiquement reconfigurables

Dans le cadre de cette thèse, l'expression "langage de modélisation" fait référence à la "syntaxe abstraite" du langage de modélisation.

Dans cette section, nous présentons le méta-modèle proposé pour assurer la modélisation des systèmes TR²E et en particulier les systèmes dynamiquement reconfigurables. Pour ce faire, nous commençons, tout d'abord, par la description du nouveau concept *MetaMode*.

3.5.1 MetaMode

Pour remédier aux problèmes soulevés par la nécessité d'énumérer toutes les configurations possibles d'un système, nous avons introduit le concept de *MetaMode* [20, 32]. Ce nouveau concept permet de caractériser les configurations au lieu de les dénombrer. Plus précisément, un *MetaMode* est défini pour un ensemble de composants structurés, pour les connexions potentielles entre ces composants ainsi que pour des contraintes structurelles et non-fonctionnelles. Par conséquent, un mode (configuration) appartenant à un *MetaMode* est défini pour un ensemble d'instances des composants structurés et des connexions logicielles du *MetaMode* respectant les contraintes structurelles et non-fonctionnelles définies dans le *MetaMode*.

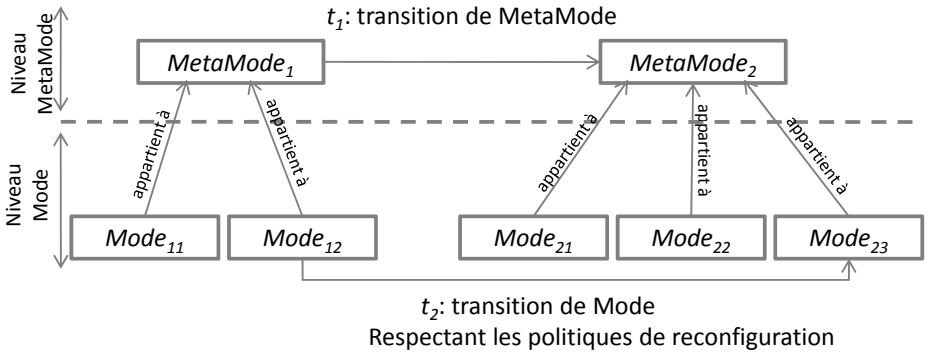

FIGURE 3.4 – Exemple illustrant les concepts *MetaMode* et *Mode*

Dans cette approche, les reconfigurations dynamiques sont spécifiées via des machines à états composées des *MetaModes* et des transitions entre ces *MetaModes*. Une transition entre deux *MetaModes* représente et caractérise un ensemble de reconfigurations entre les modes de deux *MetaModes*. Lorsqu'un événement (représenté par une transition) est déclenché, un ensemble de reconfigurations sera appliqué sur le mode courant afin d'obtenir l'un des modes appartenant au *MetaMode* cible (voir Figure 3.4). Le choix du mode cible est déterminé en se basant sur des politiques de reconfiguration.

Après la spécification des *MetaModes*, chaque *MetaMode* doit être alloué sur une instance de l'architecture matérielle du système. L'allocation est spécifiée par le mapping des modèles logiciels sur des supports d'exécution. Des contraintes d'allocation doivent être définies afin de spécifier les politiques d'allocation des modèles logiciels sur une instance matérielle. Ces contraintes d'allocation sont décrites en utilisant le langage VSL (Value Specification Language) du profil MARTE [45].

3.5.2 Exemple d'illustration : Système de gestion de charge de travail

Nous présentons dans cette sous-section un système de gestion de charge de travail (eng. Workload Manager) [7, 62] comme un exemple d'illustration. Il s'agit d'un processus périodique pouvant gérer des charges de travail variables. Les travaux réguliers sont effectués par une tâche périodique. Les travaux supplémentaires sont délégués à une tâche sporadique. Par ailleurs, le système est capable de recevoir des interruptions venant de l'extérieur. Ces interruptions sont reçues par une tâche et sont enregistrées dans un tampon spécifique. Le traitement de ces interruptions est délégué à une tâche sporadique qui est réveillée de temps en temps par la tâche

périodique principale.

Ce système a plusieurs modes de fonctionnement, la transition d'un mode à un autre s'effectue par le déclenchement d'un évènement. Pour réduire la taille de l'exemple, nous présentons dans la Figure 3.5 une machine à états présentant les reconfigurations dynamiques de ce système entre quatre modes seulement :

- Le mode *RegularWorkloadManager1* (Figure 3.6) : le système permet seulement d'accomplir les commandes régulières avec une instance du composant RegularProducer et une instance du composant *OnCallProducer*.
- Le mode *RegularWorkloadManager2* (Figure 3.7) : le système permet seulement d'accomplir les commandes régulières avec une instance du composant RegularProducer et deux instances du composant *OnCallProducer*.
- Le mode *AdvancedWorkloadManager1* (Figure 3.8) : le système permet d'accomplir les commandes régulières et de répondre aux interruptions avec une instance de chaque composant.
- Le mode *AdvancedWorkloadManager2* (Figure 3.9) : le système permet d'accomplir les commandes régulières et de répondre aux interruptions avec deux instances du composant *OnCallProducer* et une instance pour chaque autre composant.

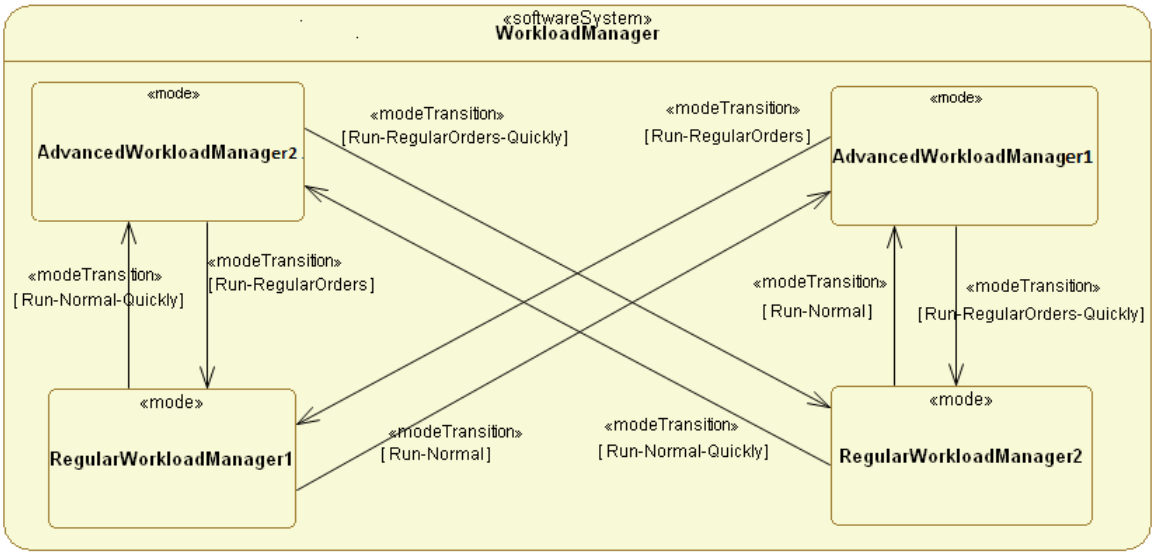

FIGURE 3.5 – Machine à état du système de gestion de travail présentant des Modes

Au lieu d'énumérer tous ces modes et afin de simplifier la spécification des transitions entre ces modes, nous proposons de définir deux *MetaModes* pour le système :

- Le *MetaMode* `RegularWorkloadManager` (Figure 3.10) : le système permet

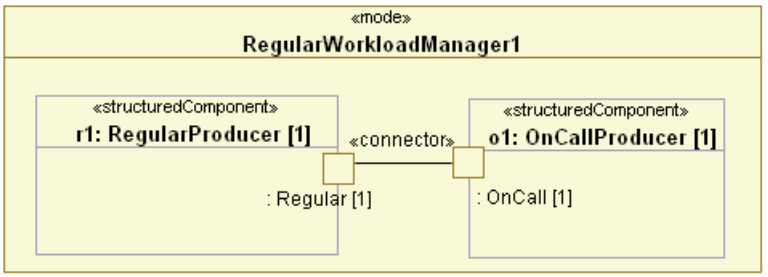

FIGURE 3.6 – Mode *RegularWorkloadManager1* du système

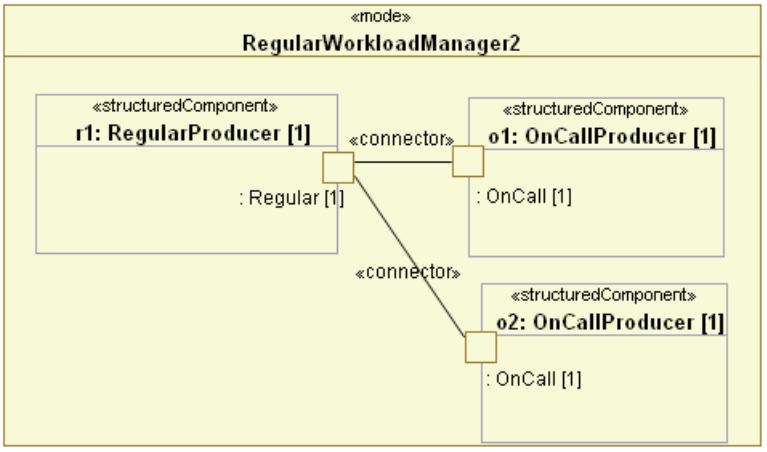

FIGURE 3.7 – Mode *RegularWorkloadManager2* du système

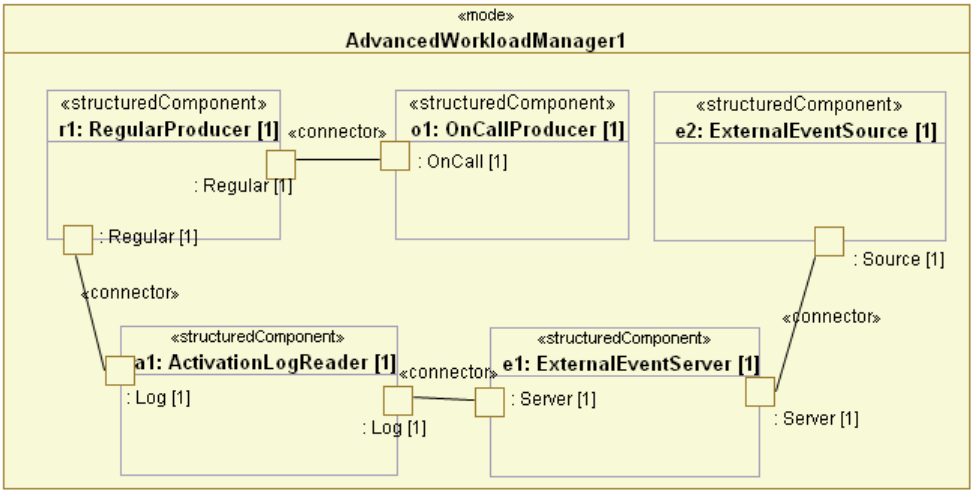

FIGURE 3.8 – Mode *AdvancedWorkloadManager1* du système

seulement d'accomplir seulement les commandes régulières. Il ne traite pas les interruptions extérieures. Ce *MetaMode* représente les deux modes *Regular-*

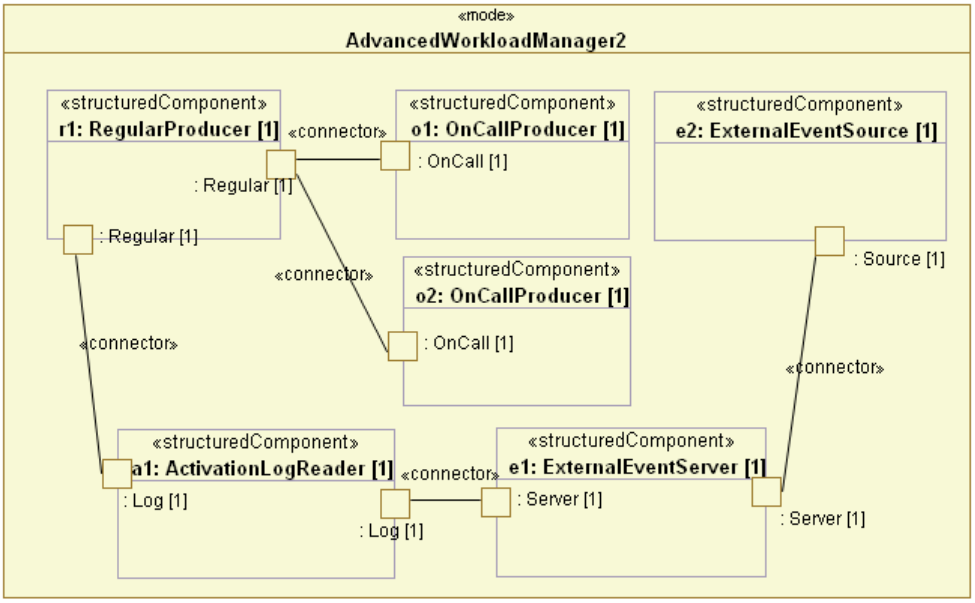

FIGURE 3.9 – Mode *AdvancedWorkloadManager2* du système

WorkloadManager1 et *RegularWorkloadManager2*.

– Le *MetaMode* `AdvancedWorkloadManager` (Figure 3.11) : le système permet d'accomplir les commandes régulières et de répondre aux interruptions. Ce *MetaMode* représente les deux modes *AdvancedWorkloadManager1* et *AdvancedWorkloadManager2*.

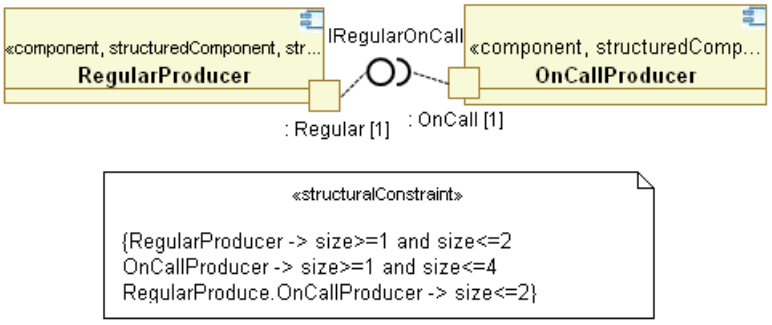

FIGURE 3.10 – MetaMode *RegularWorkloadManager* du système

La Figure 3.12 représente une nouvelle machine à états spécifiant les reconfigurations dynamiques entre les différents *MetaModes* du système. Par exemple, le passage du MetaMode "`AdvancedWorkloadManager`" vers le MetaMode "`RegulatrWorkloadManager`" se produit quand l'utilisateur exige que le système de gestion de charge de travail traite seulement les ordres réguliers.

Nous détaillons dans la suite seulement le *MetaMode*

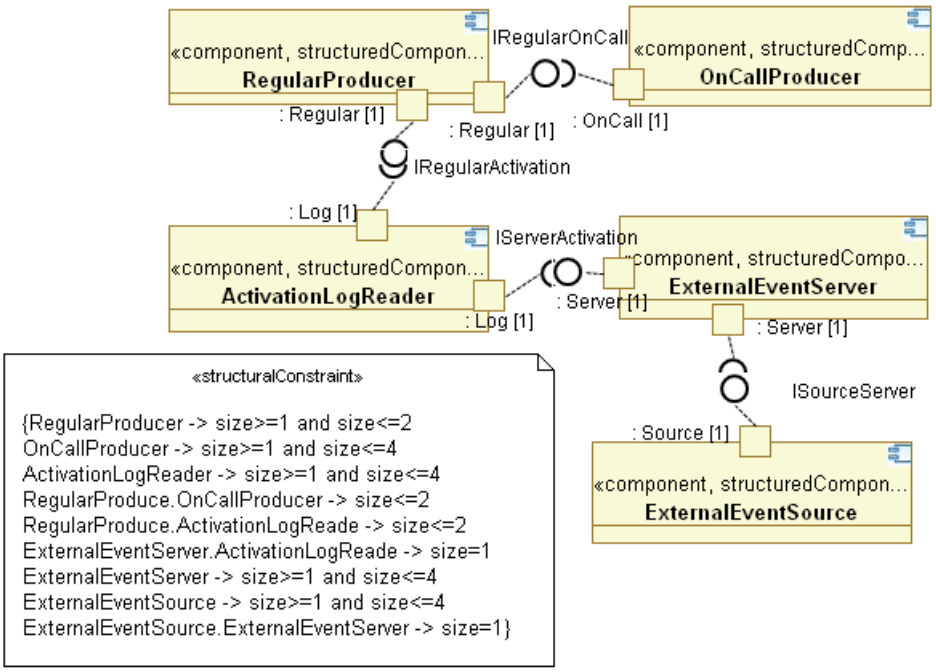

FIGURE 3.11 – MetaMode *AdvancedWorkloadManager* du système

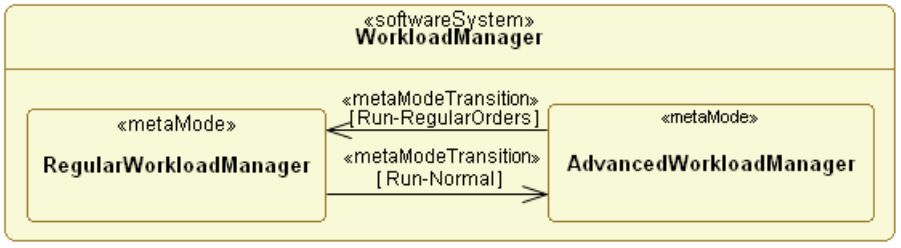

FIGURE 3.12 – Machine à états du système de gestion de charge de travail présentant les MetaModes

"`AdvancedWorkloadManager`" décrit par la Figure 3.11. Ce metaMode est défini par cinq composants (tâches) :

- Le composant *ExternalEventServer* effectue la réception des interruptions extérieures et leur enregistrement dans un tampon spécifique.
- Le composant *ExternalEventSource* simule les interruptions et les envoie d'une manière aléatoire au composant External Event Server.
- Le composant *RegularProducer* effectue des charges de travail régulières. Il délègue aussi, sous des conditions spécifiques, la charge de travail supplémentaire au composant *OnCallProducer* et le traitement des interruptions extérieures au composant *ActivationLogReader*.
- Le composant *OnCallProducer* effectue la charge de travail supplémentaire.

– Le composant *ActivationLogReader* effectue une quantité de travail qui correspond au traitement de la dernière interruption reçue par le composant *ExternalEventServer*.

Chaque composant structuré est caractérisé par un ensemble de propriétés non-fonctionnelles définies dans le tableau 3.1. De plus, le *MetaMode AdvancedWorkloadManager* devrait être alloué sur l'architecture matérielle du système (le nœud `WorkLoadManager` et le nœud `interruptionSimulator`) comme le montre la Figure 3.13.

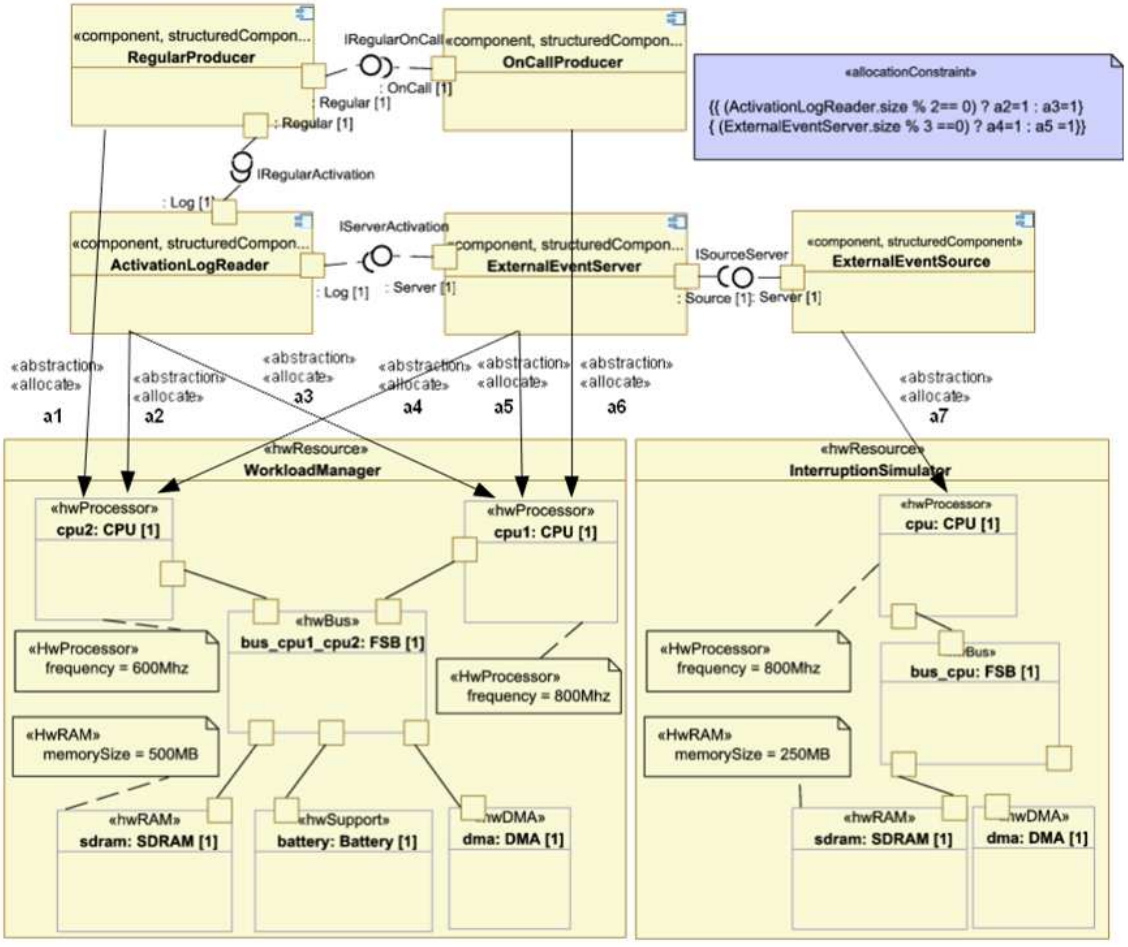

FIGURE 3.13 – Allocation du *MetaMode NotmalWorkloadManager* sur les deux noeuds *workloadManager* et *InterruptionSimulator*

3.5.3 Méta-modèle RCA4RTES

Dans cette section, nous présentons un nouveau méta-modèle permettant la spécification des systèmes TR²E dynamiquement reconfigurables [31, 32]. Ce méta-

TABLE 3.1 – Les propriétés non fonctionnelles des composants structurés [62]

Composant structurel	Nature	Period Deadline	WCET1 1Ghz	WCET2	Memory Size
Regular Producer	periodic	1000ms	498ms	2ms	0.95MB
On Call Producer	sporadic	1000ms	200ms	2ms	0.255MB
Activation Log Reader	sporadic	1000ms	125ms	4ms	0.17MB
External Event Server	sporadic	5000ms	2ms	0	0.5MB
External Event Source	sporadic	5000ms	2ms	0	0.5MB

modèle nommé RCA4RTES (Reconfigurable Component Architecture for Real-Time Embedded Systems) permet de formaliser un ensemble de nouveaux concepts pour capturer les différents aspects de la reconfiguration dynamique à différents niveaux d'abstraction. Nous utilisons ci-après le diagramme de classes d'UML pour représenter les éléments de ce méta-modèle.

La méta-classe *SoftwareSystem* a un ensemble de *MetaModes* et des transitions entre ses *MetaModes*. Une transition représentée par la méta-classe *MetaModeTransition* permet de passer d'un *MetaMode* à un autre lorsqu'un événement est déclenché. Un événement représenté par la méta-classe *MetaModeChangeEvent* peut être produit soit par l'application, soit par l'environnement comme il est décrit par l'énumération *MetaModeChangeEventKind*. Une transition au niveau MetaMode représente une abstraction de transitions au niveau Mode. Elle sera concrétisée avec un algorithme pour changer le mode courant par un autre mode.

Les politiques de reconfiguration proposées (consommation CPU, mémoire et bande passante) sont définies comme des propriétés pour la méta-classe *SoftwareSystem*. Chaque propriété définit un taux maximal de consommation des ressources matérielles (mémoires, processeurs et bus). Les modes de chaque MetaMode sont ceux ne dépassant pas les taux de consommation définis.

Le méta-modèle RCA4RTES introduit aussi la méta-classe *MetaMode* qui est composée d'un ensemble de composants structurés, d'un ensemble de connexions et de contraintes structurelles et non-fonctionnelles. Pour cela, nous définissons la méta-classe *StructuredComponent* composée d'un ensemble de ports d'interaction. Chaque composant structuré peut être une tâche périodique, sporadique, apériodique (l'énumération *DispatchProtocolKind*) ou une composition de tâches. Une connexion,

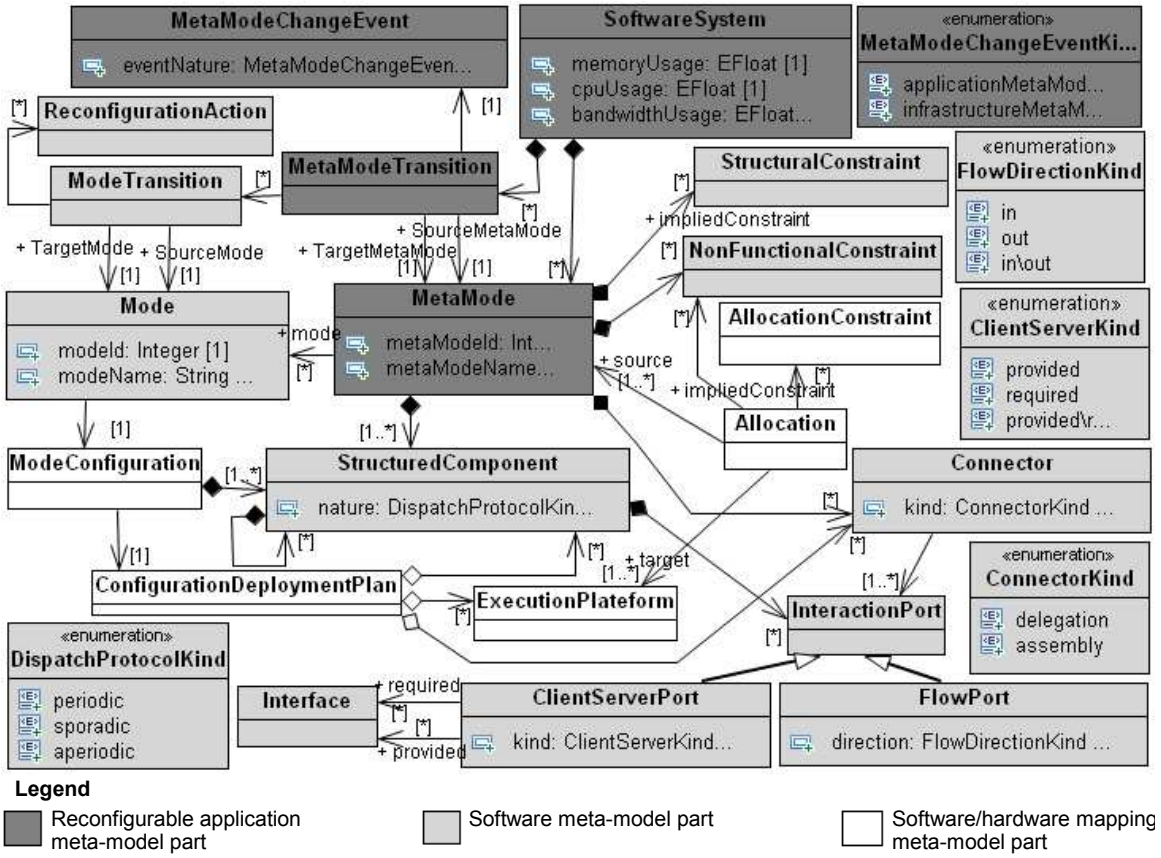

FIGURE 3.14 – Méta-modèle RCA4RTES

représentée par la méta-classe *Connector*, relie deux ou plusieurs ports d'interaction. Une connexion peut être une connexion d'assemblage ou une connexion de délégation comme l'indique l'énumération *ConnectorKind*. Nous définissons deux types de ports : les ports de flux et les ports client/serveur. Les ports de flux, définis par la méta-classe *FlowPort*, ont chacun une direction typée par l'énumération *FlowDirectionKind* (port d'entrée, port de sortie ou port d'entée/sortie). Les ports client/serveur définis par la méta-classe *ClientServerPort* peuvent avoir des interfaces fournies ou requises.

Pour spécifier les différents *MetaModes*, le méta-modèle décrit les trois types de contraintes suivants :

– Les contraintes structurelles : sont liées à la structure d'une architecture à base de composants et représentées par la méta-classe *StruturalConstraint*. Par exemple, nous pouvons définir des contraintes sur le nombre d'instances d'un composant *Receiver* comme suit :
Receiver → *size>0 and Receiver* → *size<4*

- Les contraintes non-fonctionnelles : définissent des conditions sur les propriétés non-fonctionnelles associées aux modèles. Elles sont représentées par la méta-classe *NonFunctionalConstraint*. Par exemple, nous pouvons ajuster la fréquence de processeur indépendamment de la charge de travail.
  {procUtiliz > (90, percent)? clockFreq==(60, MHz) : clockFreq==(20, MHz)}
- Les contraintes d'allocation : identifient les politiques d'allocation des modèles logiciels (*MetaModes*) sur une instance matérielle. Elles sont représentées par la méta-classe *AllocationConstraint*.
  La Figure 3.15 montre un exemple de contrainte d'allocation en utilisant le langage VSL. Nous observons que chaque nouvelle instance créée du composant *SoftwareComponent* doit être allouée sur le `cpu1` si le numéro de composant est un nombre pair, sinon elle sera allouée sur le `cpu2`.

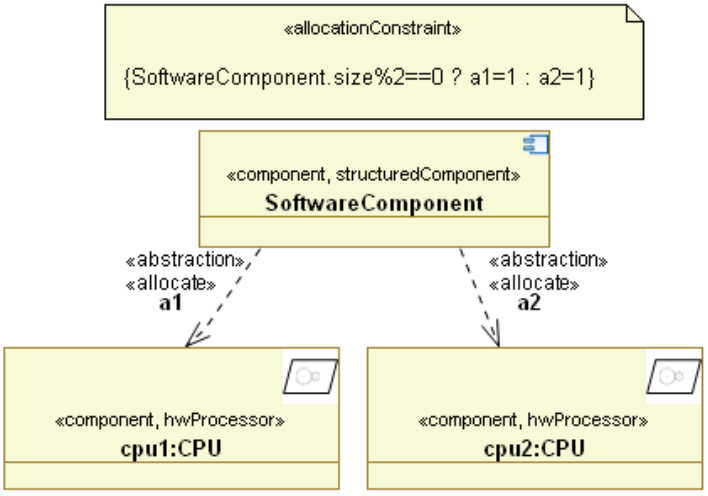

FIGURE 3.15 – Exemple d'une contrainte d'allocation

Chaque MetaMode a plusieurs modes qui sont représentés par la méta-classe *Mode*. La méta-classe *ModeConfiguration* relie la méta-classe *Mode* à la méta-classe *ConfigurationDeploymentPlan*. Cette dernière décrit une configuration par un ensemble de composants structurés, un ensemble de connexions, leurs configurations et leurs allocations sur des noeuds matériels.

Pour modéliser l'allocation, nous introduisons la méta-classe *Allocation* qui permet de spécifier l'allocation des MetaModes sur des supports d'exécution (l'allocation des modèles logiciels sur une instance matérielle) avec divers types de contraintes non-fonctionnelles et d'allocation.

3.6 Conclusion

Dans ce chapitre, nous avons présenté le processus de développement proposé pour aider à la spécification des systèmes TR^{2}E dynamiquement reconfigurables. A cet effet, de nouveaux concepts permettant de spécifier ces systèmes sont proposés dans un méta-modèle nommé RCA4RTES. Ce méta-modèle permet de capturer les reconfigurations, les architectures logicielles et matérielles, les contraintes structurelles et non-fonctionnelles de ces systèmes à un haut niveau d'abstraction. Ces contraintes sont à vérifier au niveau modèle pour produire des modèles conformes aux exigences des architectures logicielles, matérielles et des politiques de reconfiguration.

Dans les chapitres suivants, nous décrivons le framework de vérification, la plateforme d'exécution et les générateurs associés. Ensuite, nous présenterons la mise en oeuvre de notre approche proposée sous forme de langage de modélisation et d'outils pour supporter les différentes phases et étapes du processus de développement des applications TR^{2}E dynamiquement reconfigurables.

Chapitre 4

Vérification des propriétés non-fonctionnelles

4.1 Introduction

Dans le chapitre précédent, nous avons présenté le processus de développement des systèmes TR^{2}E dynamiquement reconfigurables proposé. Nous avons souligné les différentes phases de ce processus. Nous avons par la suite décrit la phase de modélisation définie dans ce processus.

Dans ce chapitre, nous nous intéressons à étudier la phase de vérification des propriétés non-fonctionnelles liées aux systèmes TR^{2}E dynamiquement reconfigurables. Pour ce faire, nous commençons d'abord par définir un ensemble de types de propriétés (section 4.2) que nous allons détailler dans les sections suivantes de ce chapitre. La section 4.3 est consacrée à la consommation CPU et au respect des échéances des tâches. Les sections 4.4 et 4.5 abordent respectivement la consommation mémoire et la consommation de la bande passante. Ensuite, nous abordons les propriétés d'absence d'interblocage et de famine (section 4.6). Finalement, nous montrons la vérification pour l'exemple d'illustration su système de gestion de charge de travail avant de conclure.

4.2 Vérification des propriétés non-fonctionnelles au niveau modèle de conception

Dans cette section, nous présentons les principes et fondements du framework de vérification [29, 33] représentant ainsi la phase de vérification de notre processus de

développement, illustré dans la Figure 3.1. Nous rappelons que le but de ce framework est la vérification des propriétés non-fonctionnelles pour des systèmes TR^{2}E dynamiquement reconfigurables au niveau modèle de conception. Pour cela, nous vérifions la consommation CPU, le respect des échéances des tâches, la consommation mémoire, la consommation de bande passante, l'absence d'interblocage et l'absence de famine. De ce fait, si ces propriétés sont satisfaites, nous pouvons passer à la phase de génération. Le cas échéant, les modèles RCA4RTES doivent être rectifiés [28].

La vérification consiste à démontrer que ces propriétés sont préservées pour les configurations du système. Pour cela, nous vérifions ces propriétés pour chaque *MetaMode* du système sans considérer les transitions d'un *MetaMode* à un autre. Comme un *MetaMode* a un ensemble de modes, il fallait donc vérifier ces propriétés pour chaque mode. Toutefois, il est souvent difficile de prévoir le nombre de modes possibles. Ainsi, nous proposons de vérifier chaque propriété pour l'instance (mode) présentant le pire cas d'exécution lié à la propriété correspondante, notée *WCEI* (*Worst Case Execution Instance*), de chaque *MetaMode*. Le respect d'une propriété pour la WCEI d'un *MetaMode* implique son respect pour tous les modes de ce *MetaMode*.

Pour chaque propriété, la WCEI de chaque *MetaMode* est déduite à partir des modèles de l'application. Dans ce cadre, nous avons proposé et développé un algorithme permettant de vérifier la spécification et de déterminer les WCEIs à vérifier. Si les modèles sont insuffisants pour obtenir les différentes WCEIs (par exemple lorsque des contraintes structurelles sont indéfinies), une intervention du concepteur est requise. En fait, notre algorithme demande au concepteur de corriger sa spécification.

Dans ce qui suit, nous détaillons les principes utilisés pour la vérification de ces propriétés non-fonctionnelles. Pour ce faire, nous considérons trois types de tâches :

- Tâche périodique : se caractérise par un intervalle de temps constant entre deux activations successives. Elle est définie par trois paramètres : une échéance (D_p), une période (P_p) et un pire temps d'exécution (C_p).
- Tâche sporadique : peut être activée par un événement à un instant aléatoire mais elle se caractérise par un délai minimal entre deux activations successives (P_{sp}). Elle est aussi définie par une échéance (D_{sp}) et un pire temps d'exécution (C_{sp}).
- Tâche apériodique : généralement activée par l'arrivée des événements (message ou requête de l'opérateur) qui peuvent se produire à tout instant. Dans

notre cas, nous allons supposer que l'événement est temporel. Elle est donc caractérisée par une date d'arrivée et un pire temps d'exécution C_{ap}).

4.3 Consommation CPU et respect des échéances des tâches

Pour vérifier la consommation CPU et le respect des échéances des tâches, nous utilisons le framework Cheddar [56] (voir sous-section 2.4.1) et l'algorithme d'ordonnancement RMS [39] (voir section 2.3).

En effet, la consommation CPU et le respect des échéances des tâches ont la même WCEI qui est définie par :

- Un nombre maximal d'instances de chaque composant structuré (chaque CPU sera alloué par un nombre maximal de tâches).
- Le pire cas d'exécution des tâches sporadiques (deux invocations consécutives de chaque tâche sporadique sont séparées par le délai minimal correspondant).

La vérification de la consommation CPU consiste ainsi à comparer le facteur d'utilisation du processeur à une borne bien déterminée (selon l'algorithme d'ordonnancement utilisé). En se basant sur l'algorithme d'ordonnancement RMS, la formule 4.1 issue de [39] doit être vérifiée pour chaque processeur du système.

$$\sum_{i=0}^{n}(C_i/P_i) \leq n(2^{1/n} - 1) \tag{4.1}$$

n : nombre de tâches périodiques

C_i : pire temps d'exécution de la tâche i.

P_i : période de la tâche i.

Le framework Cheddar est utilisé pour la simulation d'une méta-période qui représente le $PPCM$ (Plus Petit Commun Multiple) des périodes des tâches périodiques (c'est-à-dire, après chaque méta-période, le système se retrouve dans son état initial). Avec des analyses spécifiques et pour un algorithme d'ordonnancement donné, Cheddar vérifie le respect des échéances des tâches et la consommation CPU, puis indique les tâches qui ne respectent pas leurs échéances. Si ces deux propriétés sont vérifiées pour la méta-période, alors elles sont vérifiées durant toute l'exécution du système. Cependant, Cheddar et RMS ne traitent que des tâches périodiques. Ainsi, pour vérifier ces deux propriétés pour les tâches sporadiques et apériodiques,

nous proposons de considérer ces tâches comme étant des tâches périodiques.

Comme nous analysons l'instance logicielle (mode) dans le pire cas d'exécution (WCEI), cela revient à examiner le pire cas d'exécution des tâches sporadiques. Celui-ci correspond à la situation où ces tâches se déclenchent le plus souvent possible (deux invocations consécutives d'une tâche sporadique sont séparées par un délai minimal). De ce fait, nous pouvons considérer une tâche sporadique comme étant une tâche périodique avec C_p=C_{sp}, P_p=P_{sp} et D_p=D_{sp}.

De même, chaque tâche apériodique est considérée comme une tâche périodique de période P. Dans le cas où il existe des tâches périodiques et/ou sporadiques, P représente le plus petit multiple de la méta-période [1] incluant le début et la fin de toutes les tâches apériodiques. S'il n'existe que des tâches apériodiques, P sera le temps de fin de la dernière tâche apériodique exécutée.

4.4 Consommation mémoire

Pour la consommation mémoire, la WCEI est caractérisée par le nombre maximal d'instances des composants structurés (une consommation maximale des mémoires).

Chaque composant structuré a la propriété *memorySize*. Cette propriété représente la taille de la pile d'une tâche (l'empreinte mémoire d'une tâche). Il convient de vérifier à chaque instant t, que la consommation mémoire de toutes les tâches en cours d'exécution sur le même noeud est inférieure à la taille de la mémoire du noeud. Pour cela, nous avons élaboré un algorithme qui permet de vérifier la consommation mémoire pour les WCEIs des *MetaModes* du système. Pour une WCEI donnée, l'algorithme traite tous les noeuds du système en se basant sur les types des tâches allouées. Nous distinguons les cas suivants :

1. Des tâches périodiques et/ou sporadiques : puisque l'allocation de la mémoire par ces types de tâches est permanente durant l'exécution du système, la taille de la mémoire consommée est alors la somme des empreintes mémoires de ces tâches. Si cette somme dépasse la taille de la mémoire du noeud, la vérification échoue, en avertissant d'un débordement de la mémoire.
2. Des tâches apériodiques : comme premier test, nous calculons la taille de la mémoire consommée par ces tâches de la même manière que le cas précédent, en considérant les tâches apériodiques comme des tâches périodiques (la somme

1. la méta-période est le PPCM des périodes des tâches périodiques et des délais minimaux des tâches sporadiques

des empreintes mémoires de ces tâches). Si la somme calculée ne dépasse pas la taille mémoire du noeud, alors la vérification de la consommation mémoire est réussie. Le cas échéant, nous détectons le chevauchement des intervalles de temps des tâches apériodiques comme décrit dans la Figure 4.1. En fait, l'allocation mémoire pour une tâche apériodique n'est effective que lorsqu'elle est activée. Pour cela, nous avons créé des intervalles élémentaires selon les dates de début et de fin des tâches apériodiques. Ensuite, nous prenons la plus grande somme des empreintes mémoires dans ces intervalles. Si la valeur maximale calculée ne dépasse pas la taille mémoire du noeud, alors le test de la consommation mémoire est réussi.

3. Des tâches périodiques et/ou sporadiques et des tâches apériodiques : dans ce cas, nous calculons la somme des deux valeurs calculées précédemment (la somme des empreintes mémoires des tâches périodiques et/ou sporadiques et la valeur maximale de la taille mémoire allouée par les tâches apériodiques). Si cette somme ne dépasse pas la taille de la mémoire du noeud alors le test de la consommation mémoire est réussi.

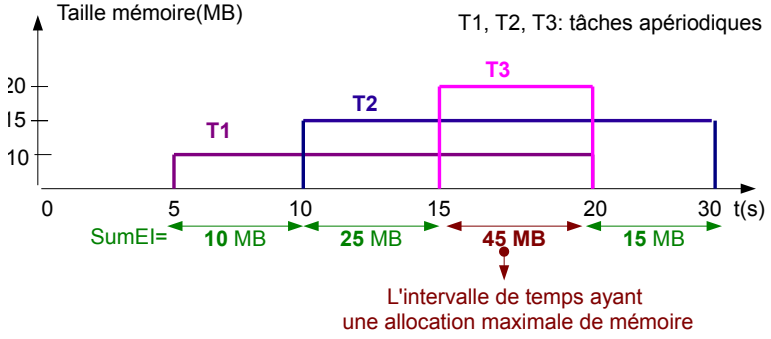

FIGURE 4.1 – Intervalles de temps des tâches apériodiques

4.5 Consommation de la bande passante

La WCEI de la propriété consommation de la bande passante est caractérisée par :

- Un nombre maximal de connexions logicielles
- L'exécution simultanée (en même temps) de toutes les connexions logicielles projetées sur le bus.

Un système embarqué distribué est composé de différents noeuds connectés entre eux par des bus. Chaque noeud peut avoir un ou plusieurs CPUs, connectés en

interne également par des bus. Il est nécessaire de vérifier que ces bus garantissent, durant l'exécution du système, l'échange des données entre les tâches, sans causer un débordement de la bande passante. Dans ce contexte, nous avons proposé un algorithme qui permet la vérification des bandes passantes de tous les bus du système intra noeud (reliant les CPUs du même noeud) et inter noeuds.

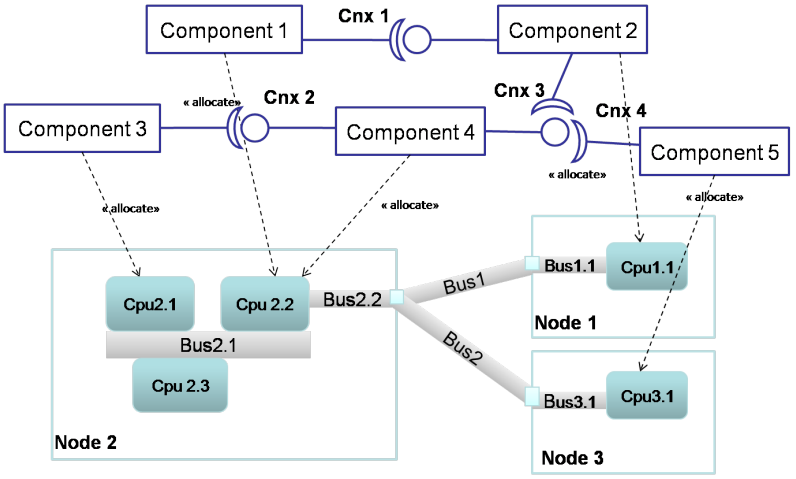

FIGURE 4.2 – Allocation de la partie logicielle sur la partie matérielle

Pour chaque noeud, nous testons l'existence d'un lien physique (bus) pour chaque connexion logicielle locale liant deux composants alloués sur deux CPUs différents. Ces deux CPUs doivent être connectés par un bus. La Figure 4.2 montre un exemple pour illustrer ce cas : le *Bus2.1* relie les deux processeurs *CPU2.1* et *CPU2.2*. Afin de tester cette connexion pour une WCEI, nous considérons le cas où toutes les connexions sont exécutées en même temps. Ainsi, la somme des bandes passantes de toutes les connexions logicielles projetées sur ce bus, ne doit pas dépasser la bande passante de ce dernier.

```
1  SumBW=0
2  For each couple of cpu connected by the bus do
3   For each software connection type i do
4    sumBw = sumBw + (BWcnxType(i)*nbMaxInstances(i))
5    if (sumBw > BwBus)) then
6            Exit ('bandwidth_overflow')
7    EndIf
8   EndFor
9  EndFor
```

Listing 4.1 – Procédure de vérification de la bande passante d'un Bus

Dans le Listing 4.1, nous présentons une procédure pour la vérification de la bande passante pour un bus donné. Pour calculer la bande passante d'un type de connexion logicielle projetée sur un bus, nous multiplions sa bande passante (`BWcnxType`) par le nombre maximal de connexions possibles (de type correspondant) pouvant être projetées sur le bus (`typenbMaxInstances`). Puisqu'un bus peut relier plus de deux CPUs (par exemple, le Bus2.1 de la Figure 4.2), nous calculons la somme des bandes passantes de tous les types de connexions possibles entre chaque couple de CPUs. La valeur calculée (`SumBW`) sera alors comparée à la bande passante du bus (`BwBus`) pour vérifier l'absence de débordement. Cette procédure est appliquée pour chaque bus utilisé pour les connexions intra-noeud.

Par la suite, nous traitons les bus utilisés pour les connexions inter noeuds du système. Nous débutons par la vérification de l'existence des bus pour chaque connexion logicielle distante liant deux composants alloués sur deux processeurs appartenant à deux noeuds différents. Donc, chaque CPU doit être connecté au port du noeud par un bus (*Bus2.2*), comme le montre la Figure 4.2, et les deux noeuds doivent être aussi connectés par un bus (*Bus1*) liant deux noeuds, comme le montre la Figure 4.2. Ensuite, nous vérifions la consommation de la bande passante de ces bus de la même manière que pour les connexions intra noeud. Pour les bus inter noeuds, nous vérifions que la somme des bandes passantes de toutes les connexions distantes projetées sur chaque bus ne dépasse pas la bande passante du bus. Pour les bus reliant un CPU par le port de noeuds, nous vérifions alors que la somme des bandes passantes de toutes les connexions logicielles locales et distantes projetées sur ce bus ne dépasse pas sa bande passante.

4.6 Absence d'interblocage et de famine

Dans cette section, nous étudions deux propriétés liées à la sûreté et à la vivacité :

- Absence d'interblocage (deadlock freedom) : le système ne se bloque pas dans une situation où plusieurs processus sont en attente les uns par rapport aux autres.
- Absence de famine (livelock freedom) : le système ne doit pas atteindre une situation dans laquelle les processus travaillent mais n'effectuent aucun avancement.

Pour vérifier ces deux propriétés dans un système, il fallait aussi vérifier toutes les configurations (modes) de chaque *MetaMode* du système. Cela est ardu puisque

il n'y a pas d'énumération de toutes les configurations possibles. De plus, nous ne pouvons pas trouver, pour ces deux propriétés, de configuration correspondant au pire cas d'exécution d'un MetaMode (la WCEI).

De ce fait et en vue d'assurer un bon fonctionnement des systèmes TR^{2}E dynamiquement reconfigurables, nous proposons de garantir ces deux propriétés par construction. Ainsi, nous définissons des restrictions qui doivent être respectées par les développeurs de ces systèmes. Ces restrictions sont inspirées du profil Ravenscar [6].

Historiquement, le profil Ravenscar présente des restrictions qui limitent l'usage des langages Ada et Java pour garantir une analyse statique d'ordonnancement et l'absence d'interblocage et de famine. Ces restrictions peuvent être étendues pour les autres langages de programmation. Pour garantir une analysabilité statique des tâches selon RMA (Rate Monotonic Analysis), Ravenscar recommande l'utilisation des tâches périodiques et sporadiques. Il interdit l'utilisation des tâches qui peuvent être déclenchées aléatoirement. Pour garantir l'absence d'interblocage et de famine, Ravenscar recommande une communication asynchrone entre les tâches via des objets partagés. Ces objets doivent être protégés contre les accès concurrents de façon à ce que deux tâches ne puissent pas accéder à un objet simultanément. Ainsi un problème de blocage peut survenir lors de l'attente des tâches pour accéder aux objets partagés. Pour remédier à ce problème, Ravenscar impose l'utilisation du protocole PCP (Priority Ceilling Protocol) [55].

Dans notre travail, nous traitons trois types de tâches : périodique, sporadique et apériodique [2]. De plus, la communication entre les tâches est asynchrone et assurée par des objets partagés en utilisant le protocole PCP.

Toutefois, les restrictions du profil Ravenscar sont établies et prouvées pour des applications centralisées. Pour les systèmes distribués utilisant des couches de transport non fiables, la construction et l'envoi des messages sont non déterministes. Ceci peut éventuellement engendrer la perte de messages. Pour cela, nous proposons l'utilisation d'une couche de transport fiable et déterministe comme par exemple SpaceWire [13]. Ainsi, nous pouvons considérer les systèmes distribués comme des systèmes monolithiques.

2. Dans notre cas, les tâches apériodiques possèdent une date d'arrivée

4.7 Vérification de l'exemple d'illustration

Après la modélisation du système de gestion de charge de travail dans la sous-section 3.5.2, nous vérifions l'ensemble des propriétés non-fonctionnelles au niveau modèle de conception. Pour cela, nous avons intégré le simulateur Cheddar. Les modèles du système obtenus seront enregistrés dans un fichier XMI qui sera l'entrée à notre framework de vérification.

Les deux Figures 4.3 et 4.4 qui présentent des sorties du framework Cheddar montrent que la consommation CPU et le respect des échéances des tâches sont bien vérifiées pour le *MetaMode* "`Insecure Workload Manager`".

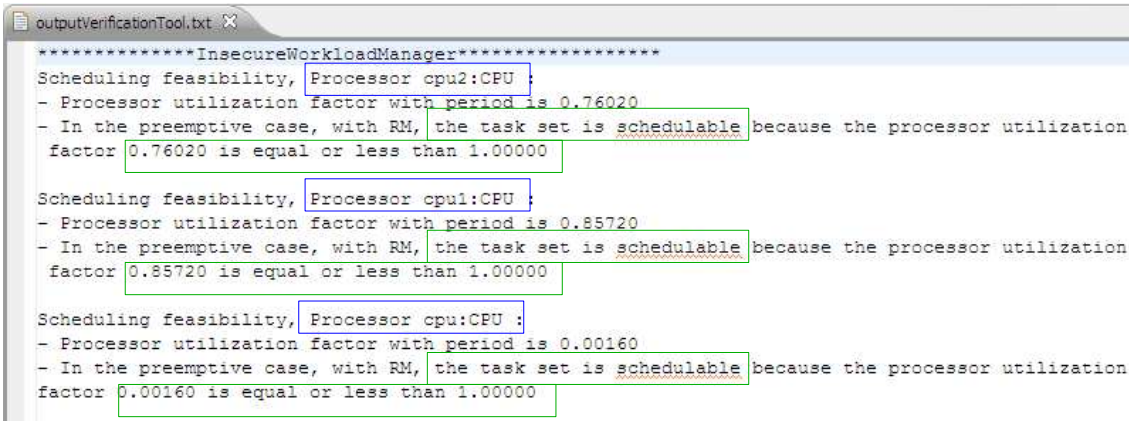

FIGURE 4.3 – Consommation CPU de WCEI du *MetaMode* Insecure Workload Manager

La consommation mémoire et la consommation de bande passante sont aussi bien vérifiées pour le *MetaMode* "`Insecure Workload Manager`". La Figure 4.5 montre que la consommation mémoire est bien vérifiée pour le *MetaMode* "`Insecure Workload Manager`" mais elle n'est pas respectée pour le *MetaMode* "`Secure Workload Manager`". La Figure 4.6 montre que la consommation de bande passante est bien vérifiée.

4.8 Conclusion

Dans ce chapitre, nous avons étudié la vérification des propriétés non fonctionnelles des systèmes TR²E dynamiquement reconfigurables au niveau modèle de conception. A cet issue, nous avons identifié l'ensemble des propriétés à prendre en considération et puis nous avons proposé des mécanismes permettant de les vérifier.

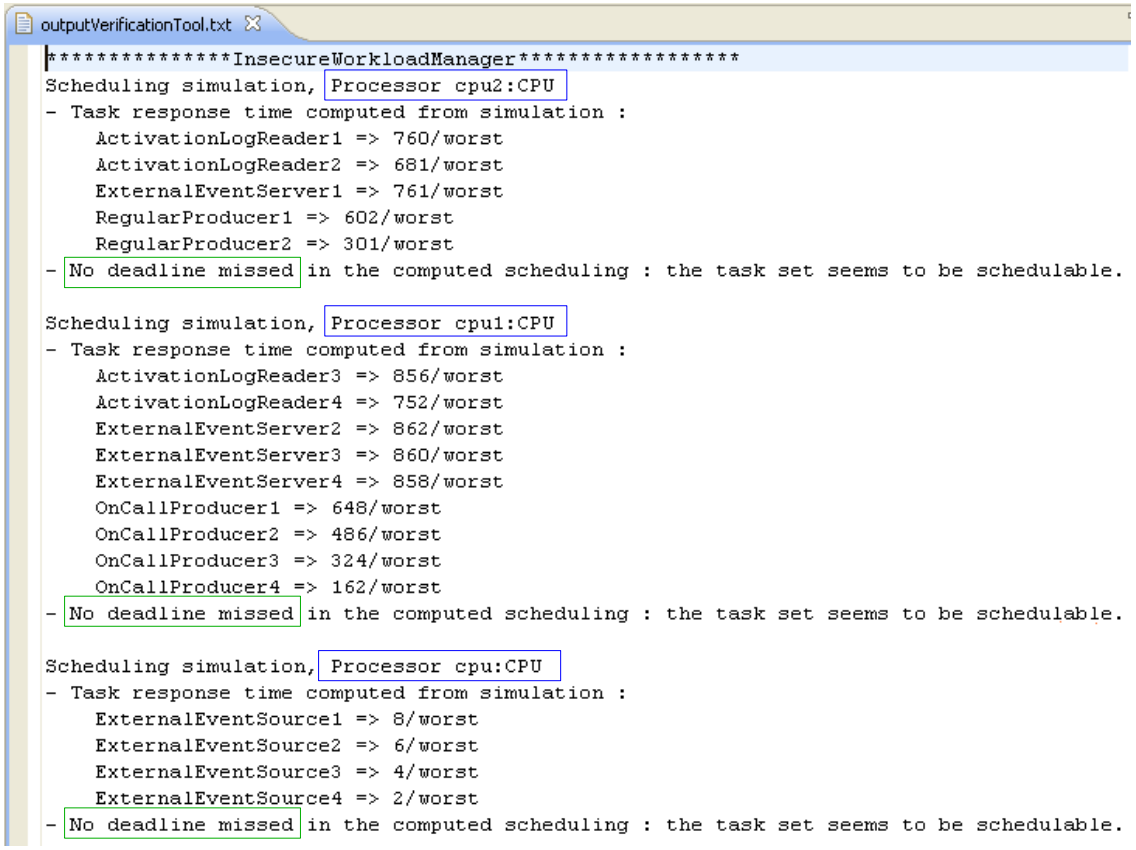

FIGURE 4.4 – Respect des deadlines de tâches de WCEI du *MetaMode* Insecure Workload Manager

Le framework de vérification défini devra être associé au framework de modélisation durant le processus de développement proposé. En effet, le framework de modélisation devra supporter la spécification de ces propriétés pour qu'elles soient par la suite vérifiées par le framework de vérification. Si l'une de ces propriétés est non respectée, le processus de modélisation est alors invoqué pour rectifier la modélisation du système.

Dans le chapitre suivant, nous aborderons en détail la plate-forme d'exécution et les générateurs associés, ce qui correspond en fait à la dernière phase de notre processus de développement (chapitre 3).

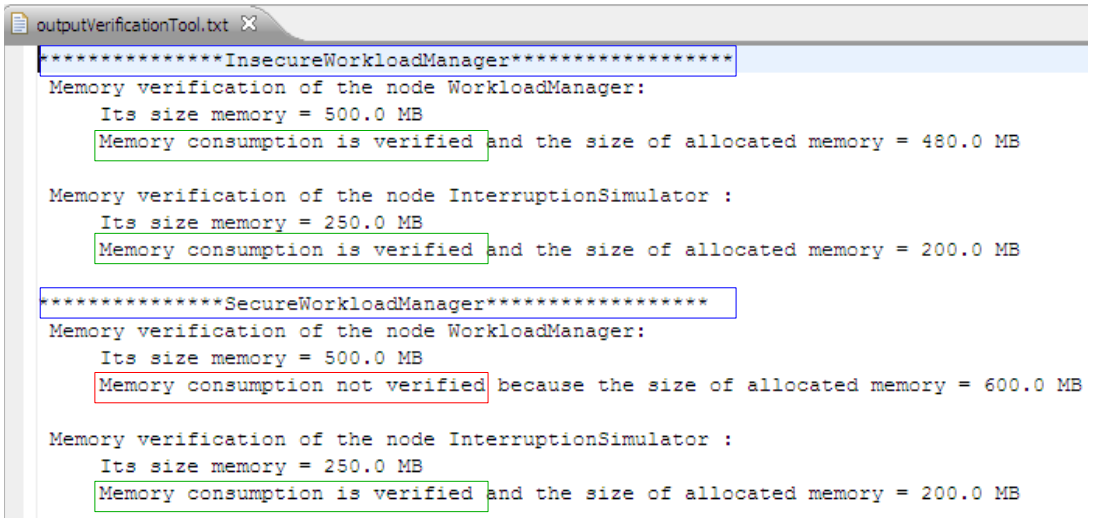

FIGURE 4.5 – Consommation mémoire des WCEIs des *MetaModes*

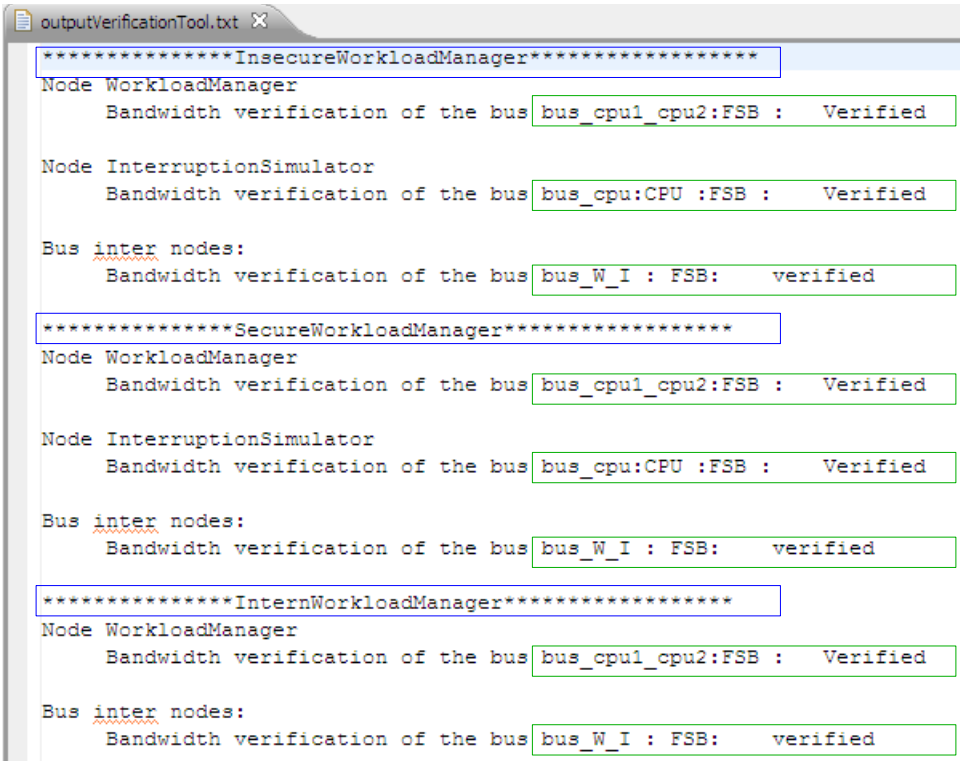

FIGURE 4.6 – Consommation de la bande passante des WCEIs des *MetaModes*

Chapitre 5

Plate-forme d'exécution des systèmes TR^{2}E

5.1 Introduction

A la base de la description détaillée des phases de modélisation et de vérification proposées dans notre processus de développement des systèmes TR^{2}E dynamiquement reconfigurables, nous nous concentrons dans ce chapitre à présenter le modèle d'une plate-forme d'exécution que nous avons développé pour ces systèmes. Cette plateforme est une extension de l'intergiciel PolyORB_HI.

En ce sens, nous commençons par la description de l'intergiciel PolyORB_HI (section 5.2), de ses caractéristiques, de ses fonctionnalités et de son utilisation pour les systèmes TR^{2}E. Ensuite et au niveau de la section 5.3, nous listons les exigences, les fonctionnalités et les fondements considérés de notre plate-forme. Nous abordons dans la section 5.4 les propriétés de la plate-forme d'exécution. Dans la section 5.5, nous démontrons la génération des modes ainsi que la génération des modèles d'implantation du système pour supporter la phase de génération du processus de développement (illustré dans la Figure 3.1). Enfin, nous conclurons le chapitre.

5.2 PolyORB_HI

PolyORB_HI [63] est un intergiciel dédié aux systèmes TR^{2}E (comme décrit dans 2.5.5). Il est inspiré de l'architecture schizophrène et il introduit des services intergiciels canoniques pour implanter la communication entre plusieurs plates-formes hétérogènes. En outre, il supporte la séparation des préoccupations. Ces services

offrent des mécanismes de communication permettant de gérer la transmission de l'information entre les nœuds d'une application répartie. Nous citons ci-dessous les services canoniques de l'intergiciel PolyORB_HI.

- **Le service adressage :** il s'occupe de la gestion des références des entités externes à l'intergiciel,
- **Le service activation :** il s'occupe de l'activation de l'entité concrète nécessaire pour traiter une requête reçue par une entité réceptrice,
- **Le service liaison :** il permet la construction des ressources de communication requises pour dialoguer avec une entité distante ou locale,
- **Le service typage :** il s'occupe de la gestion des types de données transmises entre les nœuds et fournies à l'application,
- **Le service exécution :** il supporte l'exécution du traitement requis par la réception d'un message. Il se situe au dessus du service d'activation et il utilise l'entité concrète activée par le service d'activation pour exécuter le traitement d'un message reçu,
- **Le service représentation :** il assure la transmission correcte des données à travers le réseau. Pour cela, il contrôle l'emballage des ressources, dans un tampon de communication, par l'émetteur et le déballage de ces ressources par le récepteur,
- **Le service interaction :** il s'occupe de la gestion de l'interaction entre deux nœuds logiques. Pour cela, il existe plusieurs modes d'interaction (par exemple le mode synchrone et le mode asynchrone),
- **Le service protocole :** il fournit les étapes nécessaires pour la transmission d'un message entre deux entités,
- **Le service transport :** il assure l'ouverture, la fermeture et l'utilisation des canaux de communication pour la transmission d'un message.

5.2.1 Personnalisation des services canoniques

Pour pouvoir utiliser les services canoniques d'un intergiciel pour une application donnée sans altérer leurs propriétés, ces services sont classés en deux familles : faiblement personnalisables ou fortement personnalisables.

Les services faiblement personnalisables sont ceux indépendants de l'application (i.e. ils sont les mêmes pour toutes les applications). Ils constituent alors l'intergiciel minimal PolyORB_HI.

Alors que les services fortement personnalisables sont ceux qui dépendent de

TABLE 5.1 – Répartition des services canoniques de PoplyORB_HI [61]

Services canoniques	Nouveaux services	Classement
Adressage	Adressage	Fortement personnalisable
Liaison	Liaison	Fortement personnalisable
Activation	Activation	Fortement personnalisable
Typage	Typage	Fortement personnalisable
Exécution	Parallélisme	Faiblement personnalisable
	Exécution	Fortement personnalisable
Représentation	Représentation élémentaire	Faiblement personnalisable
	Représentation avancée	Fortement personnalisable
Interaction	Interrogation	Faiblement personnalisable
	Interaction	Fortement personnalisable
Protocole	Protocole	Faiblement personnalisable
Transport	Couche basse de transport	Faiblement personnalisable
	Couche haute de transport	Fortement personnalisable

l'application. De ce fait, ils sont paramétrables en fonction des besoins de l'application cible. Le tableau 5.1 illustre la répartition des services canoniques selon leur degré de personnalisation [61]. Chaque service composé de deux parties faiblement personnalisable et fortement personnalisable est divisé en deux nouveaux services.

L'adressage, la liaison, l'activation et le typage sont des services canoniques fortement personnalisables. Ils seront alors implantés avec le code fonctionnel de l'application. Le protocole représente un service canonique faiblement personnalisable. Il sera alors fourni par l'intergiciel PolyORB_HI. L'exécution, la représentation, l'interaction et le transport, sont des services canoniques qui sont à la fois faiblement personnalisables et fortement personnalisables. L'exécution peut se décomposer en deux sous services : le service parallélisme qui est faiblement personnalisable et le service exécution qui est fortement personnalisable. Le service représentation est constitué d'un service représentation élémentaire faiblement personnalisable et d'un autre représentation avancée fortement personnalisable. De même, le service interaction est composé d'un service interrogation faiblement personnalisable et d'un autre interaction fortement personnalisable. Pour le service transport, il offre un service faiblement personnalisable pour la couche basse de transport et un service fortement personnalisable pour la couche haute de transport.

5.2.2 Support des constructions de systèmes TR²E

Pour respecter les exigences des constructions des systèmes TR²E, PolyORB_HI prend en charge les constructions clefs suivantes :

- **Les processus légers :** l'intergiciel PolyORB_HI fournit des mécanismes permettant de créer les différentes tâches supportées par les systèmes critiques comme par exemple les tâches périodiques et les tâches sporadiques.
- **Les éléments d'interface :** cet intergiciel offre aussi des mécanismes permettant aux différentes entités d'une application répartie ainsi qu'au code fourni par l'utilisateur d'effectuer de façon déterministe les opérations de lecture et d'écriture au niveau des interfaces des processus légers,
- **Les données partagées :** PolyORB_HI garantit la protection des données partagées entre les différentes entités de l'application si cela n'est pas traité par le langage de programmation. Les mécanismes de protection fournis sont à base de routines de verrouillage et de déverrouillage pour des accès cohérents aux données partagées.

5.2.3 Faible empreinte mémoire

La dernière caractéristique de l'intergiciel PolyORB_HI est sa faible empreinte mémoire. En effet, dans chaque nœud, PolyORB_HI contient seulement les services canoniques faiblement personnalisables. Les composants fortement personnalisables sont implantés avec le code fonctionnel de l'application selon les caractéristiques de l'application.

5.3 Intergiciel dédié pour les systèmes TR²E dynamiquement reconfigurables

Afin d'adresser les limites identifiées dans les intergiciels étudiés et présentés dans le chapitre 2, nous proposons un intergiciel [35], nommé RCES4RTES (Reconfigurable Component Execution Support for Real-Time Embedded Systems), pour supporter l'exécution des systèmes TR²E dynamiquement reconfigurables. Cet intergiciel inclut des générateurs pour sa propre configuration et des générateurs du code pour aider au développement d'une grande partie du code pour ces systèmes.

5.3.1 Analyse des besoins

Nous analysons les besoins pour la réalisation de la plate-forme d'exécution pour les systèmes TR²E dynamiquement reconfigurables. L'analyse couvre les éléments

nécessaires à la modélisation de cette plate-forme, les besoins de ce système en terme d'architecture logicielle et les propriétés non-fonctionnelles.

L'intergiciel RCES4RTES doit offrir des routines pour supporter les reconfigurations dynamiques ainsi que la supervision et la cohérence. Il doit respecter les aspects temps réel avec une faible empreinte mémoire. En effet, RCES4RTES doit offrir les fonctionnalités suivantes :

- Supporter la supervision des systèmes TR²E en vérifiant au cours de l'exécution la topologie et le comportement de l'architecture, en traçant l'exécution du système [1] et en mettant à jour les variables partagées. La supervision peut être aussi utilisée pour assurer la réflexivité du système.
- Garantir la cohérence du système durant et après des reconfigurations.
- Respecter les contraintes temps réel. En effet, pour encapsuler les reconfigurations dynamiques, une tâche est automatiquement créée sur chaque noeud du système. Elle représente une tâche sporadique pour appliquer des actions de reconfiguration. Elle sera considérée comme une tâche du système et sera ordonnancée avec les autres tâches. En utilisant cette tâche sporadique, notre intergiciel gère les reconfigurations sans affecter l'exécution des tâches du système et sans dépasser leurs échéances.
- Assurer la communication entre les plates-formes hétérogènes en utilisant l'architecture schizophrène et ses services canoniques.

Comme nous allons le montrer dans la suite (section 6.5.1), l'intergiciel s'inspire des règles du profil Ravenscar [6] afin d'assurer l'ordonnancement des tâches du système en garantissant l'absence d'interblocage et de famine.

5.3.2 Modèle de conception de l'intergiciel RCES4RTES

La Figure 5.1 montre une partie du modèle de l'intergiciel RCES4RTES sous forme de diagramme de classes UML. Ce dernier supporte trois types de tâches : périodique, sporadique et apériodique représentées respectivement par les classes *PeriodicTask*, *SporadicTask* et *AperiodicTask*. Ces tâches communiquent à travers des ports d'entrée et de sortie définis respectivement par les classes *PortIn* et *PortOut*. Cette communication est assurée en utilisant des routeurs de ports et des structures de données représentés respectivement par la classe *PortRouter* et par la classe *GeneratedType*.

La classe *Event* permet d'introduire les événements du système qui déclenchent

1. Une trace exhibant le nombre de composants et de connexions

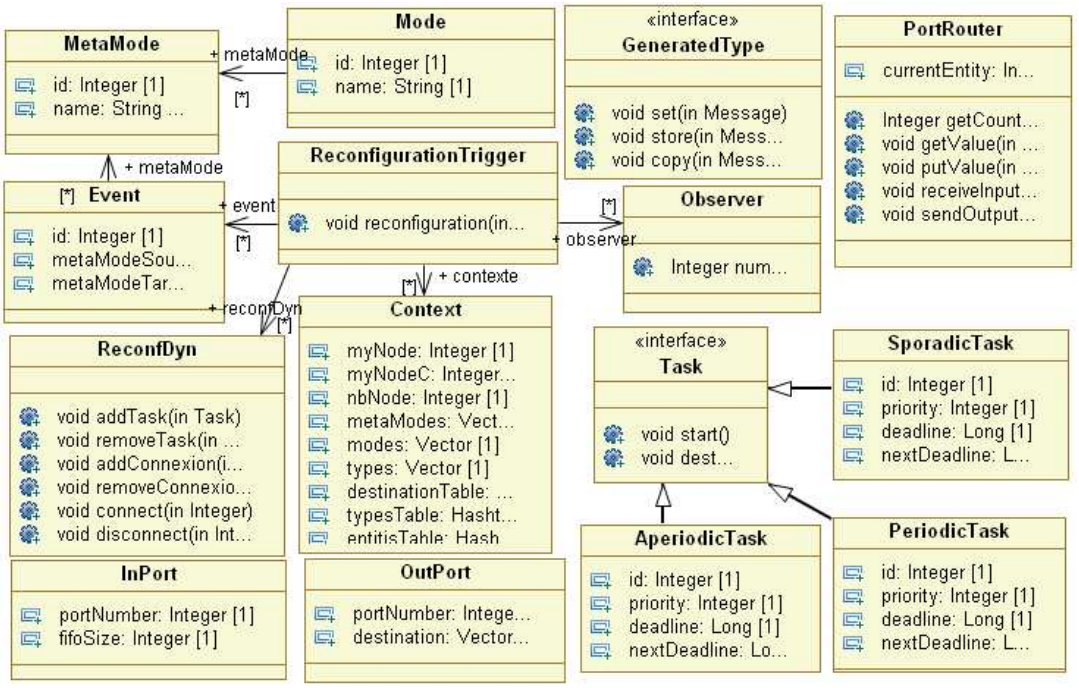

FIGURE 5.1 – Partie du modèle de l'intergiciel RCES4RTES

des actions de reconfiguration. Le déclenchement de l'un de ces événements peut changer le *MetaMode* courant du système par un autre. La classe *Mode* est utilisée par la classe *ReconfigurationTrigger* afin de déterminer (1) le mode courant ainsi que le mode cible et (2) les reconfigurations à appliquer par une simple comparaison entre les deux modes. La classe *ReconfigurationTrigger* utilise les fonctions fournies par la classe *ReconfDyn* (*addTask*, *removeTask*, *addConnexion*, *removeConnexion*, etc) pour appliquer les reconfigurations. En plus, cette classe permet de mettre à jour les attributs de la classe *Context* qui représente la topologie de l'application à chaque instant au cours de l'exécution.

5.3.3 Reconfigurations dynamiques

RCES4RTES traite deux types de reconfigurations : des reconfigurations architecturales et des reconfigurations comportementales. Les reconfigurations architecturales consistent à modifier la topologie d'une architecture logicielle, comme par exemple l'ajout et la suppression d'une connexion ou d'un composant. Les reconfigurations comportementales consistent à modifier le comportement du système, en adaptant par exemple les propriétés non-fonctionnelles du système.

RCES4RTES supporte les reconfigurations architecturales suivantes :

- Connecter des nœuds : ajouter un canal de communication entre deux noeuds,
- Déconnecter des nœuds : supprimer un canal de communication entre deux noeuds,
- Ajouter une connexion : ajouter un canal de communication entre deux composants,
- Supprimer une connexion : supprimer un canal de communication entre deux composants,
- Ajouter un composant : créer et déployer un composant sur une plate-forme d'exécution et le connecter avec les autres composants,
- Supprimer un composant : supprimer le composant et ses connexions avec les autres composants,
- Migrer un composant : déplacer un composant d'un noeud vers un autre nœud.

RCES4RTES supporte les reconfigurations comportementales suivantes :

- Mettre à jour les propriétés d'un composant : en leur affectant des nouvelles valeurs.
- Remplacer un composant : remplacer un composant par un autre.

Pour supporter ces reconfigurations et mettre à jour l'état courant de l'application en termes de noeuds, composants, connexions et ports, RCES4RTES utilise un ensemble de structures de données dans la classe *Context* :

- *myNodesC* : cette structure est définie dans chaque noeud afin de décrire les interconnexions du noeud avec les autres noeuds et gérer les interconnexions dynamiques des noeuds. Chaque paire de noeuds ayant au moins une connexion entre leurs composants doivent être connectée. Les opérations de connexion et de déconnexions des noeuds seront reportées par la mise à jour de ces structures dans les noeuds correspondants.
- *destinationTable* : pour chaque noeud, cette structure est utilisée pour représenter les ports de destination de chaque port d'un composant déployé sur ce noeud. L'ajout et la suppression des connexions seront traduites par la mise à jour de cette structure.
- *entitiesTable* : cette structure contient tous les noeuds de l'application et les instances des composants déployées sur ces noeuds. Cette structure est utilisée durant l'ajout, la suppression et la migration des composants.
- *portsTable* : cette structure de données contient les instances des composants de l'application avec leurs ports. Cette structure est utilisée durant l'ajout, la suppression et la migration des composants.

5.4 Propriétés des systèmes TR²E dynamiquement reconfigurables

En plus des routines de reconfiguration dynamique, l'intergiciel RCES4RTES permet de garantir certaines propriétés pour les systèmes TR²E dynamiquement reconfigurables. Dans cette section, nous abordons la cohérence, la supervision et les aspects temps réel.

5.4.1 Cohérence

Pour garantir la cohérence d'un système TR²E dynamiquement reconfigurable, RCES4RTES permet de gérer la perte des messages durant une reconfiguration. Chaque composant impliqué dans un processus de reconfiguration doit être bloqué durant la reconfiguration. Pour cela, RCES4RTES fournit deux routines permettant de bloquer et de débloquer un composant au cours de l'exécution du système. Le blocage d'un composant consiste à empêcher les composants sources de lui envoyer des messages et d'achever le traitement des requêtes courantes. Inversement, le déblocage d'un composant consiste à relâcher le blocage en autorisant l'envoi des requêtes par les composants sources.

En vue d'éviter toute perte de temps et de minimiser la durée de blocage, ces deux routines (i.e. bloquer et débloquer des composants) doivent être exécutées respectivement après la création des nouvelles instances des composants et avant la suppression des instances des composants dans les routines de reconfiguration.

5.4.2 Supervision

RCES4RTES fournit des routines pour la supervision d'un système TR²E et pour l'observation de son état durant son exécution. En effet, la supervision d'un système au cours de l'exécution aide à gérer les reconfigurations dynamiques. Pour cela, nous avons implanté des routines (1) pour obtenir le nombre de composants au cours de l'exécution, (2) pour obtenir le nombre de connexions entre un composant donné et les autres composants et (3) et pour obtenir le temps d'accès en mode lecture/écriture à des variables partagées.

5.4.3 Aspect temps réel

La durée d'une reconfiguration T_{rd} , comme décrite dans l'équation 5.1, est la somme de la durée de blocage des composants T_b, de la durée du transfert de l'état d'un composant T_{state} [2] et de la durée de l'exécution des actions de reconfiguration T_{act}.

Durant la période T_{act}, on exécute la tâche de reconfiguration dynamique d'un noeud en informant les autres noeuds des reconfigurations appliquées en respectant les contraintes temporelles. Elle est donc considérée comme une tâche du système et sera ordonnancée avec les autres tâches.

$$T_{rd} = T_b + T_{state} + T_{act} \tag{5.1}$$

Les durées T_b, T_{state} et T_{act} sont proportionnelles à la taille des données à traiter. T_{act} dépend aussi de la fréquence des processeurs et de la couche du transport. Afin d'avoir une durée de reconfiguration déterministe, nous proposons d'utiliser une couche de transport déterministe et fiable comme par exemple la couche de transport SpaceWire [13].

5.5 Phase de Génération

Dans cette section, nous détaillons la phase de génération dans notre processus de développement, comme illustré dans la Figure 3.1. Nous distinguons deux types de générations (voir Figure 5.2) : (1) la génération des modèles d'implémentation et (2) la génération des modes de chaque MetaMode respectant les politiques de reconfiguration.

5.5.1 Génération des modes

La génération des modes permet d'obtenir l'ensemble des modes de chaque *MetaMode* respectant les politiques de reconfiguration à partir des modèles RCA4RTES.

La génération des modes est utilisée pour configurer l'intergiciel en lui ajoutant ces modes. Une liste contenant l'ensemble des *MetaModes* du système, une liste contenant l'ensemble des événements déclenchant les reconfigurations et une liste contenant l'ensemble des modes respectant les politiques de reconfiguration seront ajoutées à la classe *Context* de l'intergiciel à partir des modèles RCA4RTES.

2. T_{state} est définie seulement dans le cas d'une migration d'un composant

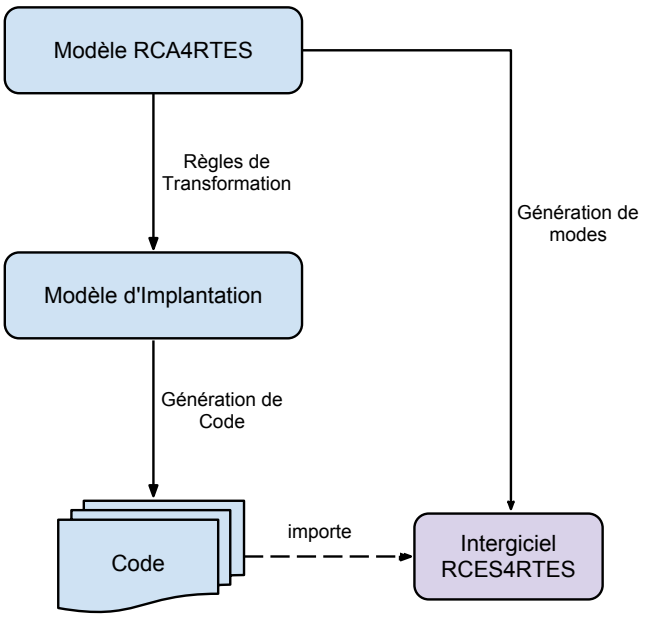

FIGURE 5.2 – Stratégie de génération de code

Les politiques de reconfiguration permettent de déterminer un seuil de consommation des ressources (mémoire, processeur, et bus) à ne pas dépasser. Ces politiques seront utilisées pour générer les modes, de chacun des MetaModes du système, respectant les taux déterminés, pour ensuite les ajouter dans l'intergiciel. Nous passons maintenant à dévoiler les trois politiques de reconfiguration.

Génération des modes respectant le taux de consommation mémoire

Pour générer les modes respectant le taux déterminé de consommation mémoire, nous allons parcourir les modes de chaque *MetaMode* du système et comparer le taux de consommation de chaque mode par rapport au taux déterminé. Pour chaque mode, il faut vérifier que le taux de consommation de chaque mémoire utilisée ne dépasse pas le taux déterminé. Pour cela, et comme le décrit la sous-section 4.4, nous distinguons trois cas :

1. Tâches périodiques et/ou sporadiques : Puisque l'allocation de la mémoire par ces deux types de tâches est permanente durant l'exécution du système, le calcul de la taille de la mémoire consommée est la somme des empreintes mémoires de ces tâches. Si le rapport entre la somme calculée de la mémoire consommée et la taille de la mémoire ne dépasse pas le taux déterminé, le mode sera alors généré.

2. Tâches apériodiques : Nous détectons le chevauchement des intervalles de temps des tâches apériodiques comme décrit dans la section 4.4. En fait, une tâche apériodique n'utilise de l'espace mémoire que lorsqu'elle est activée. Pour cela, nous avons créé des intervalles élémentaires selon les dates de début et de fin des tâches apériodiques. Ensuite, nous calculons la somme des plus grandes empreintes mémoires dans ces intervalles. Si le rapport entre la plus grande somme et la taille de la mémoire ne dépasse pas le taux déterminé, le mode sera alors généré.
3. Tâches périodiques et/ou sporadiques avec des tâches apériodiques : Dans ce cas, nous calculons la somme de deux valeurs calculées précédemment (la somme des empreintes mémoires des tâches périodiques et/ou sporadiques et la valeur maximale de la taille mémoire allouée par les tâches apériodiques). Si le rapport entre la somme calculée de la mémoire consommée et la taille mémoire ne dépasse pas le taux déterminé pour chaque mémoire, le mode sera alors généré.

Génération des modes respectant le taux de consommation CPU

Pour chaque mode, il faut vérifier que le taux de consommation de chaque CPU utilisé ne dépasse pas le taux déterminé. La vérification de la consommation CPU consiste à comparer le facteur d'utilisation du processeur au taux déterminé.

Le taux de consommation CPU par des tâches périodiques est calculé par la formule $\sum_{i=0}^{n}(Ci/Pi)$, où n est le nombre de tâches périodiques et Ci et Pi sont respectivement le pire temps d'exécution et la période de la tâche i.

Pour vérifier le taux de consommation CPU par des tâches sporadiques et apériodiques, nous proposons de considérer ces tâches comme étant des tâches périodiques comme le décrit la section 4.3 et d'utiliser la même formule que celle des tâches périodiques. Si le taux de consommation de chaque processeur du mode ne dépasse pas le taux déterminé, le mode est alors généré.

Génération des modes respectant le taux de consommation de bande passante

Pour générer les modes respectant le taux déterminé de consommation de la bande passante, il faut vérifier que le taux de consommation de chaque bus utilisé, en considérant le cas où toutes les connexions seront établies en même temps, ne dépasse pas le taux déterminé.

Nous calculons la somme des bandes passantes de toutes les connexions logicielles projetées sur un bus comme déjà décrit dans la section 4.5. Si le rapport entre la somme calculée et la bande passante du bus est inférieur au taux déterminé pour chaque bus, le mode sera généré.

5.5.2 Méta-modèle d'implantation

Pour aider à la génération du code, les modèles RCA4RTES, spécifiés dans la phase de modélisation, seront transformés vers un modèle d'implantation dédié à la plate-forme d'exécution cible (l'intergiciel RCES4RTES). Ce modèle d'implantation importe le modèle de RCES4RTES pour que le code à générer puisse utiliser les routines implantées par cet intergiciel. Nous commençons par la présentation du méta-modèle d'implantation.

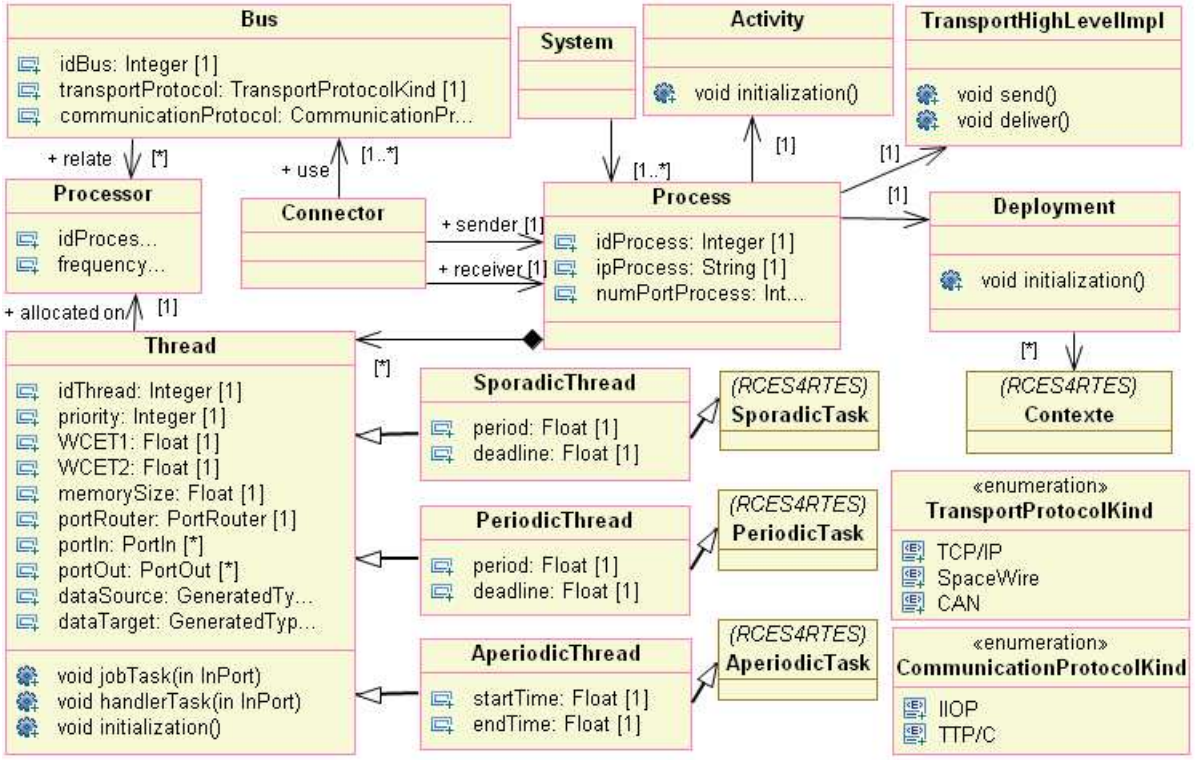

FIGURE 5.3 – Méta-modèle d'implantation

Le méta-modèle d'implantation décrit par un diagramme de classes UML dans la Figure 5.3 capture un ensemble de concepts pour modéliser des implantations de systèmes TR²E dynamiquement reconfigurables. Chaque implantation d'un tel système, conforme à ce méta-modèle, sera représentée par un ensemble de processus et de connexions.

- *Process.* Nous définissons les méta-classes *System* et *Process* pour représenter respectivement le système à modéliser et ses processus. Un processus est composé d'un ensemble de tâches définies par la méta-classe *Thread*. Ces tâches peuvent être périodiques (*PeriodicThread*), sporadiques (*SporadicThread*) ou apériodiques (*AperiodicThread*). Ces classes héritent respectivement des classes *PeriodicTask*, *SporadicTask* et *AperiodicTask* de l'intergiciel. Chaque tâche est caractérisée par un ensemble de propriétés non-fonctionnelles telles que la priorité. Elle a un ensemble de ports d'entrée et de sortie définis respectivement par les deux classes *PortIn* et *PortOut* de l'intergiciel. Ces ports permettent d'envoyer et de recevoir des données de type *GeneratedType* à travers un routeur de port défini par la classe *PortRouter* du modèle de l'intergiciel.

  La partie fonctionnelle de chaque tâche sera ajoutée par le développeur dans la méthode *threadJob* après la génération du code (section 6.6.3). Chaque tâche sera allouée sur un processeur choisi en accord avec les contraintes d'allocation spécifiées au niveau modèle de conception. La méta-classe *Processor* est caractérisée par une fréquence (propriété *frequency*). Tous les processeurs sont liés par des bus définis par la méta-classe *Bus*.
- *Connector.* La méta-classe *Connector* décrit la communication entre deux processus à travers des bus décrits par la méta-classe *Bus*.
- *Bus.* Chaque bus est caractérisé par un protocole de communication et un protocole de transport spécifiés respectivement par les deux énumérations *CommunicationProtocolKind* et *TransportProtocolKind*.
- *Deployment.* Cette méta-classe représente le déploiement du mode initial en utilisant la classe *Context* du modèle de l'intergiciel
- *TransportHighLevelImpl.* Cette méta-classe gère l'envoi et la réception des données entre les tâches,
- *Activity.* Cette méta-classe gère le lancement des tâches du système et de la tâche assurant la reconfiguration dynamique (instance de la classe *ReconfigurationTrigger* du modèle de l'intergiciel).

5.5.3 Génération du code

A partir des modèles d'implantation, nous utilisons Acceleo [3] pour la génération du code du système. Le code généré importe l'implantation de l'intergiciel

3. http ://www.eclipse.org/acceleo

RCES4RTES pour utiliser ses routines.

Par souci de pertinence et pour éviter toute redondance, nous détaillons cette section dans le chapitre 6 dans le cadre d'une extension de l'intergiciel PolyORB_HI.

5.6 Conclusion

Dans ce chapitre, nous avons présenté le modèle de la plate-forme d'exécution pour les systèmes TR²E dynamiquement reconfigurables sous forme d'un intergiciel. Cette plate-forme fournit également des générateurs de modèles d'implantation et de modes. A l'issue de ce constat, les modèles RCA4RTES spécifiés dans la phase de modélisation seront transformés vers des modèles d'implantation aidant à générer une grande partie de code en se basant sur les routines fournies par l'intergiciel proposé. L'intergiciel sera ainsi configuré grâce aux différents modes générés, respectant les politiques de reconfiguration déjà définies.

Dans le chapitre suivant, nous présentons l'outillage du processus de développement de systèmes TR²E dynamiquement reconfigurables par la définition de nouveaux outils de modélisation, de nouveaux mécanismes de vérification et d'une implantation de l'intergiciel RCES4RTES et de ses générateurs.

Chapitre 6

Outillage de l'approche

6.1 Introduction

Dans les chapitres précédents, nous avons proposé un processus de développement des systèmes TR^{2}E dynamiquement reconfigurables. Ce processus est dirigé par différentes phases allant de la modélisation à un haut niveau d'abstraction jusqu'à la génération de code pour une plate-forme d'exécution dédiée.

Dans le cadre de l'outillage de notre processus de développement, un environnement de modélisation et d'exécution ont été proposés. Il s'agit d'un ensemble de langages de modélisation et de principes de vérification. En outre, un modèle pour les plates-formes d'exécution de ces systèmes, des modèles d'implantation, des générateurs ainsi que de bonnes pratiques pour l'implantation ont été aussi présentés.

Dans ce chapitre, nous décrivons l'outillage de notre approche à l'aide d'un certain nombre d'outils. Nous entamons ce chapitre par présenter l'architecture de la suite d'outils. Ensuite, nous décrivons les outils de modélisation (section 6.3), les outils de vérification (section 6.4) et enfin la construction de l'intergiciel, sa configuration (section 6.5) et la génération de l'implémentation (section 6.6).

6.2 Architecture de la suite d'outils

Pour appliquer les différentes phases du processus de développement des systèmes TR^{2}E dynamiquement reconfigurables, nous avons développé une suite d'outils dont l'architecture est montrée dans la Figure 6.1.

Cette suite d'outils est basée sur la plate-forme Eclipse [1] et son framework de

1. http ://www.eclipse.org

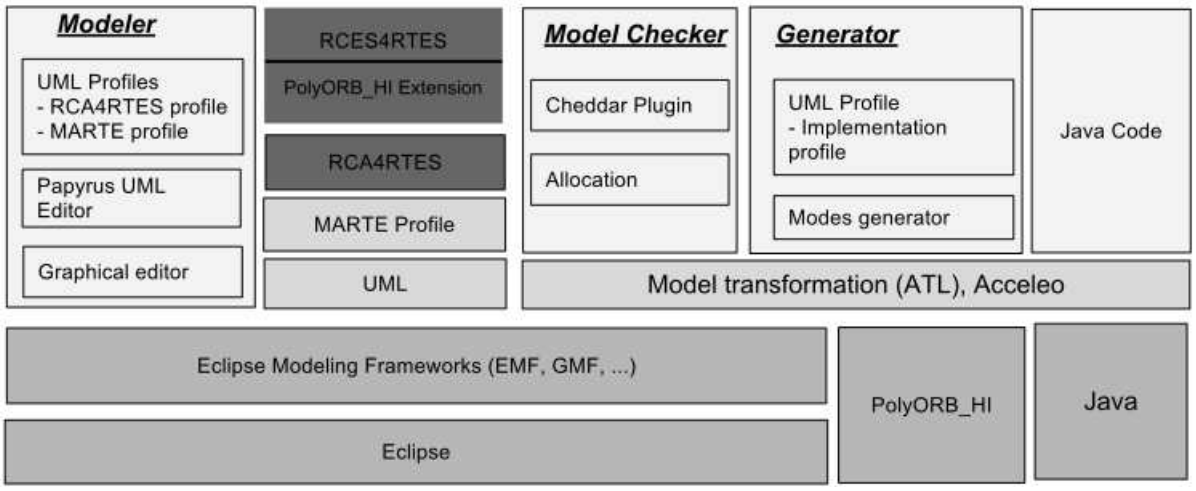

FIGURE 6.1 – Architecture de la suite d'outils

modélisation EMF [57]. Elle est composée de l'ensemble des briques suivants :

- La brique des outils de modélisation : composée de l'éditeur UML Papyrus, le profil MARTE et le profil RCA4RTES,
- La brique des outils de vérification : proposée sous forme d'un plugin Eclipse intégrant le framework Cheddar et les extensions proposées,
- La brique des outils de génération : offrant un générateur de modes des différents *MetaModes* respectant les politiques de reconfiguration spécifiées, et un ensemble de plugins pour les transformations de modèles. Un plugin à base d'ATL [2] (Atlas Transformation Language) [25] et un autre à base d'Acceleo [3] pour les transformations modèle vers modèle et modèle vers texte, respectivement.
- Une plate-forme d'exécution : représentant une implantation d'une extension de PolyORB_HI basée sur RTSJ (Real-time Specification for Java [5]).

6.3 Outils de modélisation

En vue d'arriver à bien exploiter les outils de modélisation existants, notamment UML et ses profils, nous avons choisi d'implanter le méta-modèle RCA4RTES (section 3.5) sous forme de profil UML. Ainsi, UML, le profil MARTE et le profil RCA4RTES sont supportés par le même éditeur UML : Papyrus [4]. La Figure 6.2

2. http ://www.eclipse.org/atl/
3. http ://www.eclipse.org/acceleo
4. http ://www.eclipse.org/papyrus/

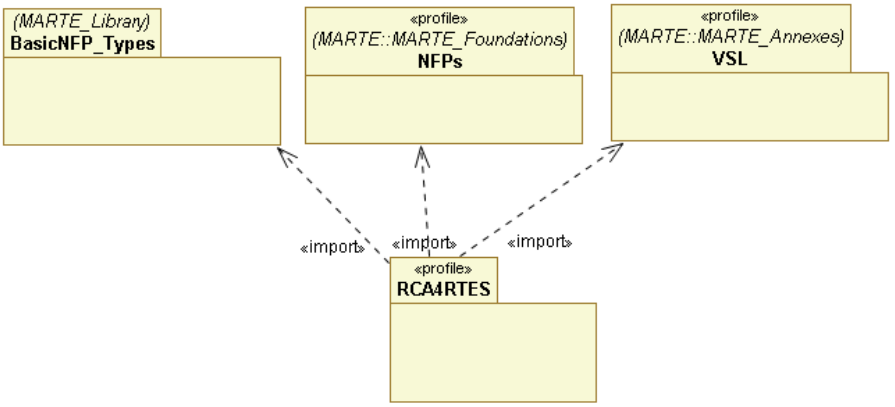

FIGURE 6.2 – Dépendances du profil UML RCA4RTES

présente les dépendances entre le profil RCA4RTES et le profil MARTE. Le profil RCA4RCES importe les deux sous profils *NFPs* et *VSL* du profil MARTE pour spécifier les contraintes non-fonctionnelles et les contraintes d'allocation. Il importe aussi la librairie des types (*Basic NFP_ Types*) du profil MARTE. Nous présentons ci-dessous en détail le profil RCA4RTES.

6.3.1 Profil RCA4RTES

Nous utilisons les profils comme un mécanisme d'extension proposé par UML, pour implanter le méta-modèle RCA4RTES. Nous présentons dans cette section les principaux concepts de ce profil, comme le montre la Figure 6.3. Nous utilisons le *template* suivant :

- **Description :** une description succincte de l'élément de modélisation,
- **Méta-classes étendues :** les extensions UML,
- **Propriétés :** les propriétés associées aux stéréotypes.
- **Généralisation :** les stéréotypes dont l'élément de modélisation hérite.

Le stéréotype *SoftwareSystem*

- **Description :** ce stéréotype permet de spécifier la machine à états présentant les reconfigurations dynamiques du système. Cette machine est composée d'un ensemble de MetaModes et de transitions entre ces MetaModes. La reconfiguration dynamique sera assujettie à un ensemble de politiques de reconfiguration pour guider la sélection du mode cible. Pour ce faire, nous introduisons un ensemble de propriétés comme des valeurs marquées pour ce stéréotype.

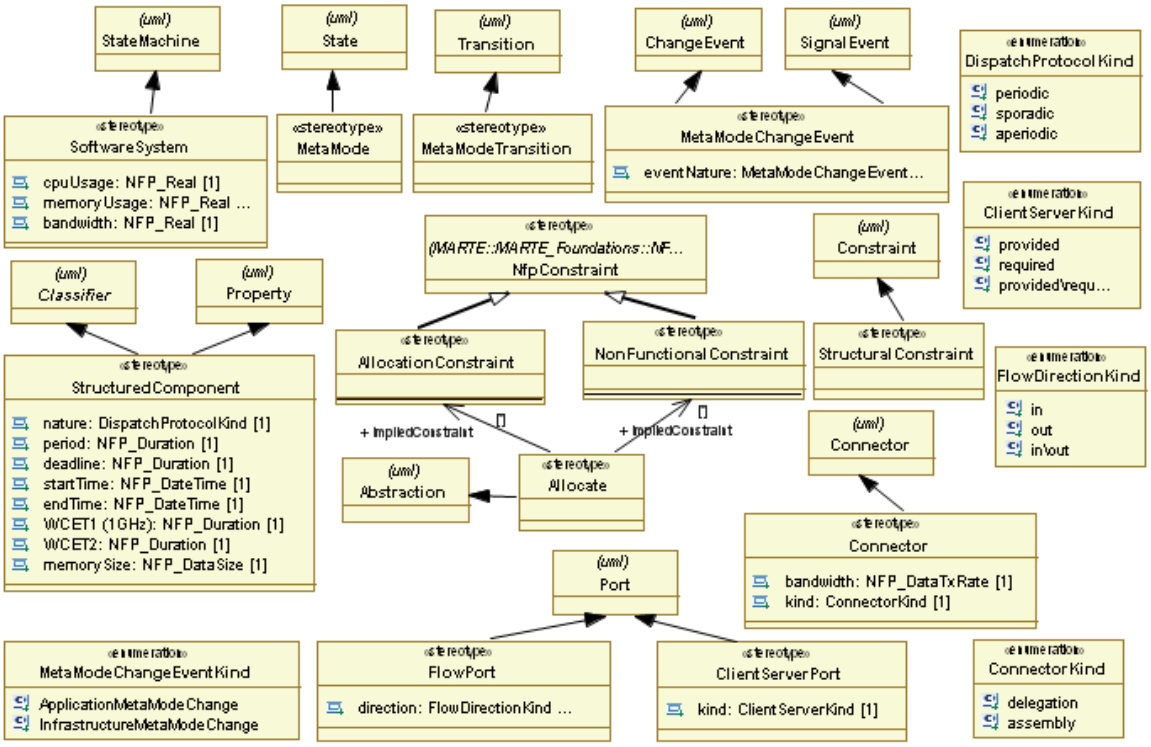

FIGURE 6.3 – Description du profil RCA4RTES

- **Méta-classe étendue :** *StateMachine*.
- **Propriétés :**
  - **memoryUsage :** elle définit le taux d'utilisation maximal de chaque mémoire. Son type est NFP_Real de la librairie des types du profil MARTE,
  - **cpuUsage :** elle définit le taux d'utilisation maximal de chaque processeur. Son type est NFP_Real de la librairie des types du profil MARTE,
  - **bandwidthUsage :** elle définit le taux d'utilisation maximum de chaque bus. Son type est NFP_Real de la librairie des types du profil MARTE.

Le stéréotype *MetaModeTransition*

- **Description :** ce stéréotype permet de spécifier les reconfigurations dynamiques entre des MetaModes par des transitons. Ces transitions seront déclenchées suite à des événements.
- **Méta-classe étendue :** *Transition*.

Le stéréotype *MetaModeChangeEvent*

- **Description :** ce stéréotype permet de spécifier l'événement provoquant les reconfigurations dynamiques entre des MetaModes.
- **Méta-classes étendues :** *ChangeEvent* et *SignalEvent*.

Le stéréotype *MetaMode*

- **Description :** ce stéréotype permet de modéliser une caractérisation d'un ensemble de configurations. Il représente l'état du système en spécifiant l'ensemble de ses composants structurés, des connexions entre ces composants ainsi que l'ensemble de contraintes non-fonctionnelles et structurelles sur cette architecture.
- **Méta-classe étendue :** *State*.

Le stéréotype *StructuredComponent*

- **Description :** ce stéréotype permet de spécifier les composants structurés pour encapsuler une tâche ou un ensemble de tâches.
- **Méta-classe étendue :** *Component*.
- **Propriétés :**
  - **nature :** elle définit la nature de la tâche qui peut être périodique, sporadique ou apériodique. Cette propriété est typée par l'énumération *DispatchProtocolKind*,
  - **period :** elle définit la période d'une tâche périodique. Elle est utilisée aussi pour définir le temps minimal entre deux activations successives d'une tâche sporadique. Cette propriété est typée par *NFP_Duration* de la librairie du profil MARTE,
  - **deadline :** elle définit l'échéance d'une tâche périodique ou sporadique. Elle est typée par *NFP_Duration* de la librairie du profil MARTE,
  - **startTime :** elle définit le temps de début d'une tâche apériodique. Elle est typée par *NFP_DateTime* de la librairie du profil MARTE,
  - **endTime :** elle définit le temps de fin d'une tâche apériodique. Elle est typée par *NFP_DateTime* de la librairie du profil MARTE,
  - **memorySize :** elle définit l'empreinte mémoire d'une tâche (la pile d'une tâche). Elle est typée par *NFP_DataSize* de la librairie du profil MARTE,
  - **WCET1** définit le pire temps d'exécution d'une tâche sur un processeur de fréquence 1 GHZ. Il est calculé par le rapport du nombre d'instructions sur

la fréquence du processeur. La valeur de WCET1 varie selon la fréquence du processeur. Elle est typée par *NFP_Duration* de la librairie du profil MARTE.

- **WCET2** définit le temps d'exécution d'une tâche qui ne dépend pas de la fréquence du processeur mais d'autres périphériques comme les bus ou les mémoires. Elle est typée par *NFP_Duration* de la librairie du profil MARTE.

Le stéréotype *FlowPort*

- **Description :** ce stéréotype permet de spécifier les ports de flux des composants structurés.
- **Méta-classe étendue :** *Port*.
- **Propriété :**
  - **direction :** cette propriété permet de spécifier la direction du flux du port. Elle est typée par l'énumération *FlowDirectionKind*. Le port peut donc être un port de sortie, un port d'entrée ou un port d'entrée/sortie de flux.

Le stéréotype *ClientServerPort*

- **Description :** ce stéréotype permet de spécifier les ports clients/serveurs des composants structurés.
- **Méta-classe étendue :** *Port*.
- **Propriété :**
  - **kind :** cette propriété permet de spécifier directement la nature du port. Elle est typée par l'énumération *ClientServerKind*. Le port peut donc être un port fourni (avoir au moins une interface fournie), requis (avoir au moins une interface requise) ou requis/fourni (avoir au moins une interface fournie et une interface requise).

Le stéréotype *Connector*

- **Description :** ce stéréotype permet de spécifier les connexions permettant de relier les composants à travers leurs ports.
- **Méta-classe étendue :** *Connector*.
- **Propriétés :**
  - **kind :** elle permet de spécifier le type de la connexion qui peut être une connexion de délégation ou une connexion d'assemblage. Cette propriété est

typée par l'énumération *ConnectorKind*.
- **bandwidth :** elle définit la bande passante de la connexion logicielle. Elle est typée par *NFP_DataTxRate* de la librairie du profil MARTE.

Le stéréotype *StructuralConstraint*

- **Description :** ce stéréotype permet de spécifier des contraintes architecturales en utilisant le langage déclaratif OCL (Object Constraint Language).
- **Méta-classe étendue :** *Constraint*.

Le stéréotype *NonFunctionalConstraint*

- **Description :** ce stéréotype permet de spécifier des contraintes non-fonctionnelles en utilisant le langage VSL (Value Specification Language) du profil MARTE.
- **Méta-classe étendue :** *Constraint*.
- **Généralisation :** le stéréotype *NfpConstraint* du sous-profil NFP du profil MARTE.

Le stéréotype *AllocationConstraint*

- **Description :** ce stéréotype permet de spécifier les politiques d'allocation en utilisant le langage VSL (Value Specification Language) du profil MARTE. Ces politiques définissent le mapping des modèles logiciels sur une instance matérielle.
- **Méta-classe étendue :** *Constraint*.
- **Généralisation :** le stéréotype *NfpConstraint* du sous-profil NFP du profil MARTE.

Le stéréotype *Allocate*

- **Description :** ce stéréotype permet de spécifier l'allocation des *MetaModes* sur les supports d'exécution. L'application de ce stéréotype peut impliquer des contraintes d'allocation et non-fonctionnelles.
- **Méta-classe étendue :** *Abstraction*.

6.3.2 Éditeur de modèles de systèmes TR²E dynamiquement reconfigurables

Nous utilisons les services offerts par l'éditeur UML Papyrus [5] pour implanter un éditeur de modèles supportant le profil UML *RCA4RTES*. Ainsi, la palette de Papyrus est étendue avec des nouveaux concepts de ce profil. La figure 6.4 montre l'environnement de modélisation. Le modèle est présenté au milieu de la figure. La palette contenant les éléments de modélisation est présentée à gauche de la figure. L'éditeur de modèles de systèmes TR²E dynamiquement reconfigurables est fourni sous forme d'un plugin Eclipse.

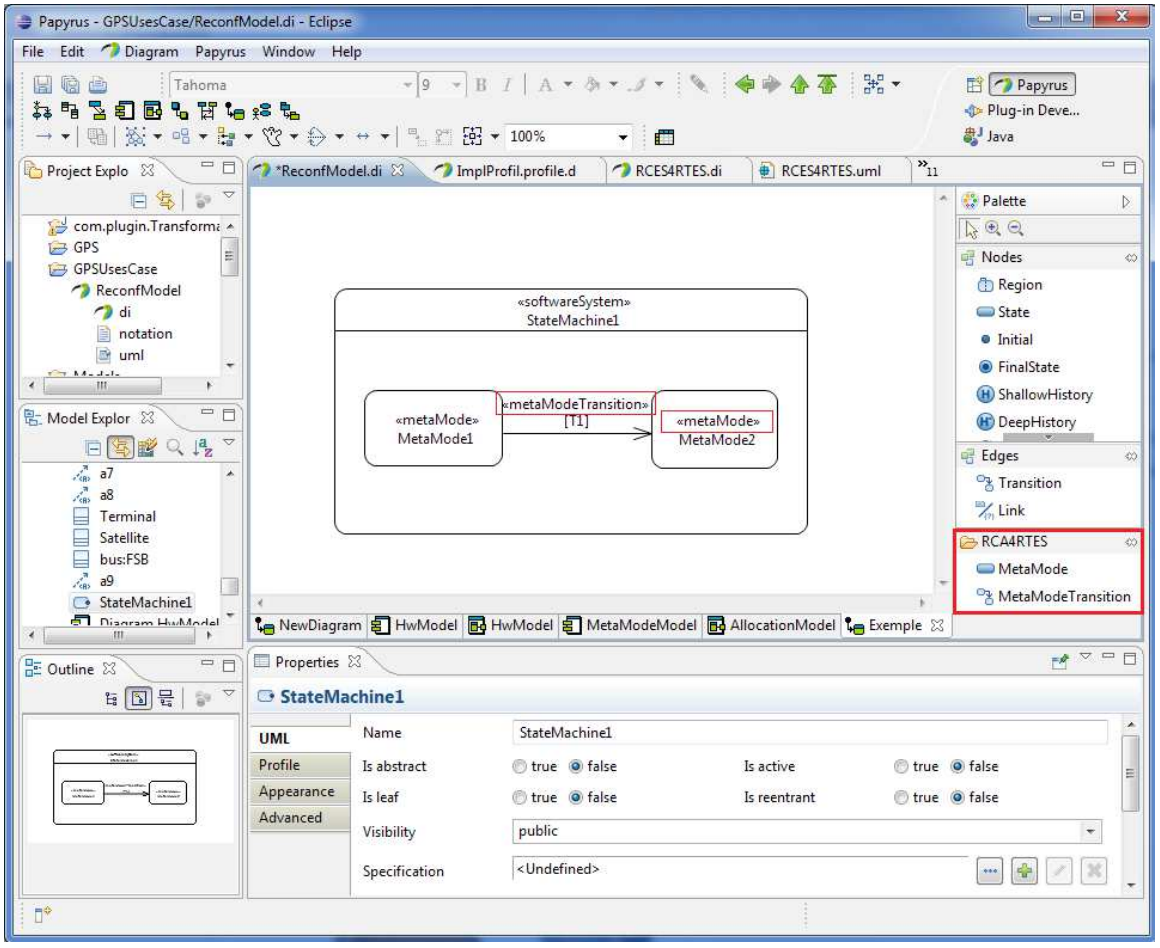

FIGURE 6.4 – Extension de la palette de l'éditeur Papyrus

5. Papyrus est un plugin Eclipse.

6.4 Vérification des systèmes TR²E

Dans cette section, nous décrivons des outils aidant à la vérification des propriétés non-fonctionnelles pour les systèmes TR²E dynamiquement reconfigurables au niveau modèle. Nous utilisons (1) le framework Cheddar (voir la sous section 2.4.1), (2) l'algorithme d'ordonnancement RMS pour la vérification de la consommation CPU et le respect des échéances des tâches, et nous définissons (3) deux nouveaux algorithmes pour la consommation mémoire et la bande passante. Par contre, les propriétés de l'absence d'interblocage et de famine sont assurées par construction [6].

6.4.1 Plugin de vérification

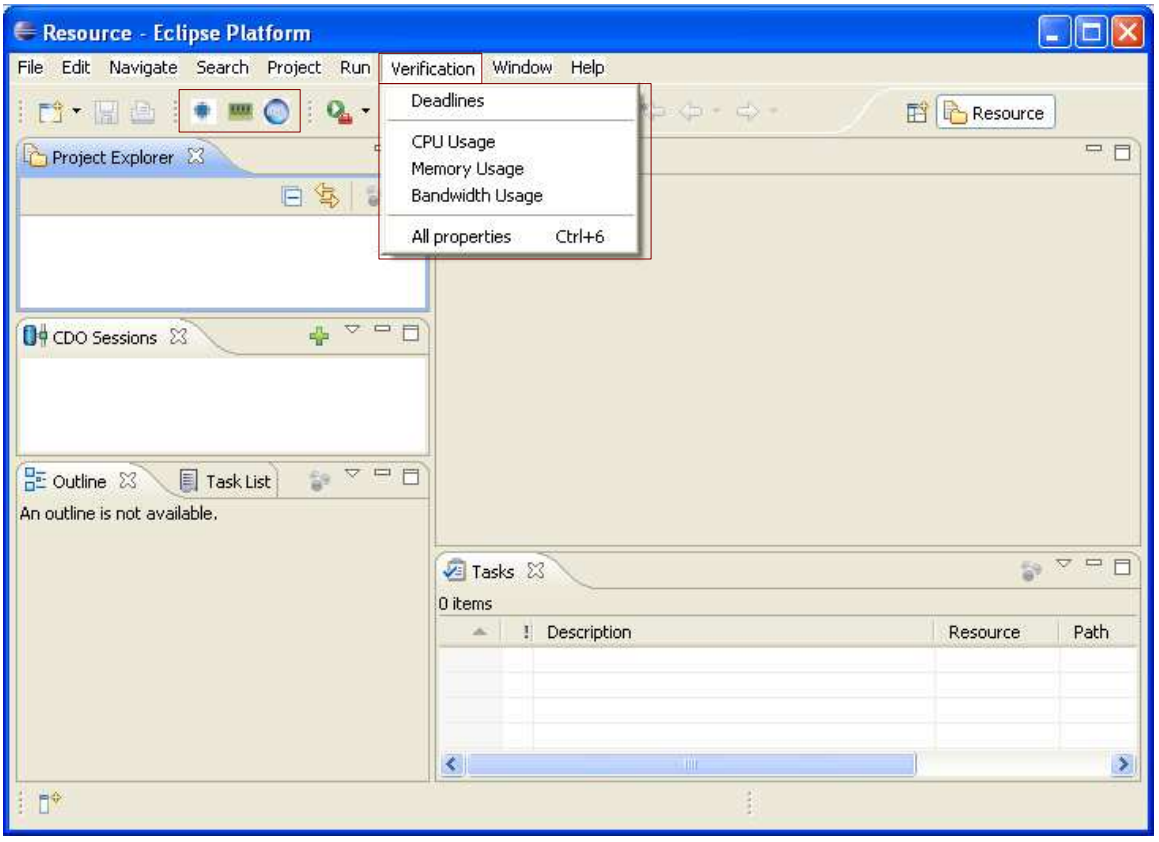

FIGURE 6.5 – Plugin Eclipse de vérification

Nous avons développé un plugin Eclipse pour la vérification de certaines propriétés non-fonctionnelles. Ce plugin permet de vérifier ces propriétés spécifiés dans le modèle décrit en UML étendu par le profil MARTE et par le profil *RCA4RTES*. Le

6. Respect du profil Ravenscar lors de l'implantation de l'intergiciel

plugin de vérification intègre le framework Cheddar afin de vérifier la consommation CPU et le respect des échéances de tâches. Il permet aussi de vérifier la consommation mémoire et la consommation de la bande passante. La figure 6.5 montre le plugin de vérification des propriétés non-fonctionnelles.

Nous montrons dans la figure 6.6 le format du fichier d'entrée du framework Cheddar. Pour notre exemple, le fichier regroupe des informations sur l'ensemble des processeurs du système ainsi que les tâches périodiques allouées sur ces processeurs. Les résultats de la vérification (sortie du framework) seront produits dans un fichier texte comme le montre la figure 6.7.

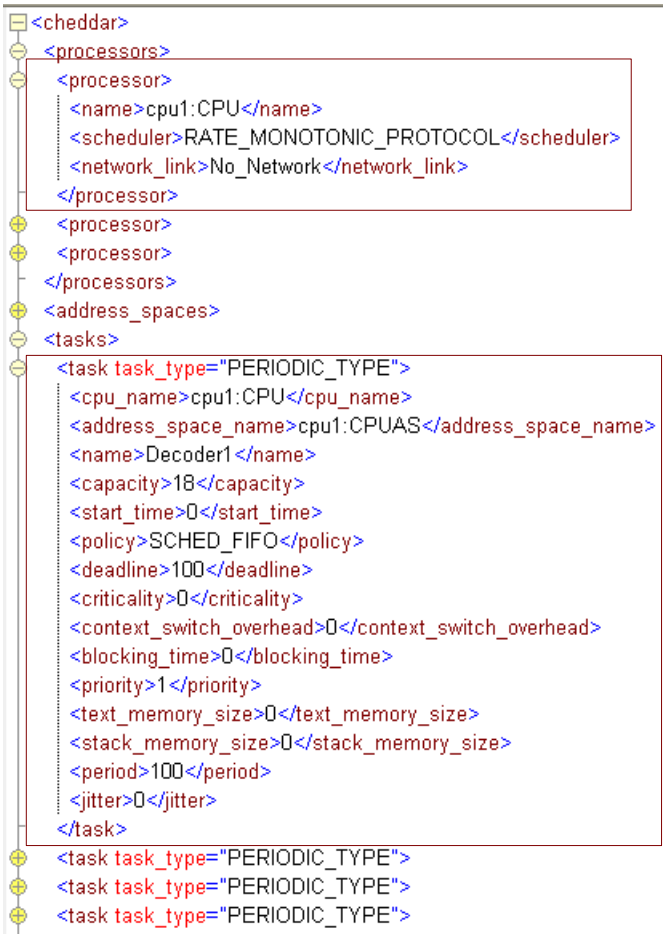

FIGURE 6.6 – Fichier XML présentant l'entrée du framework Cheddar

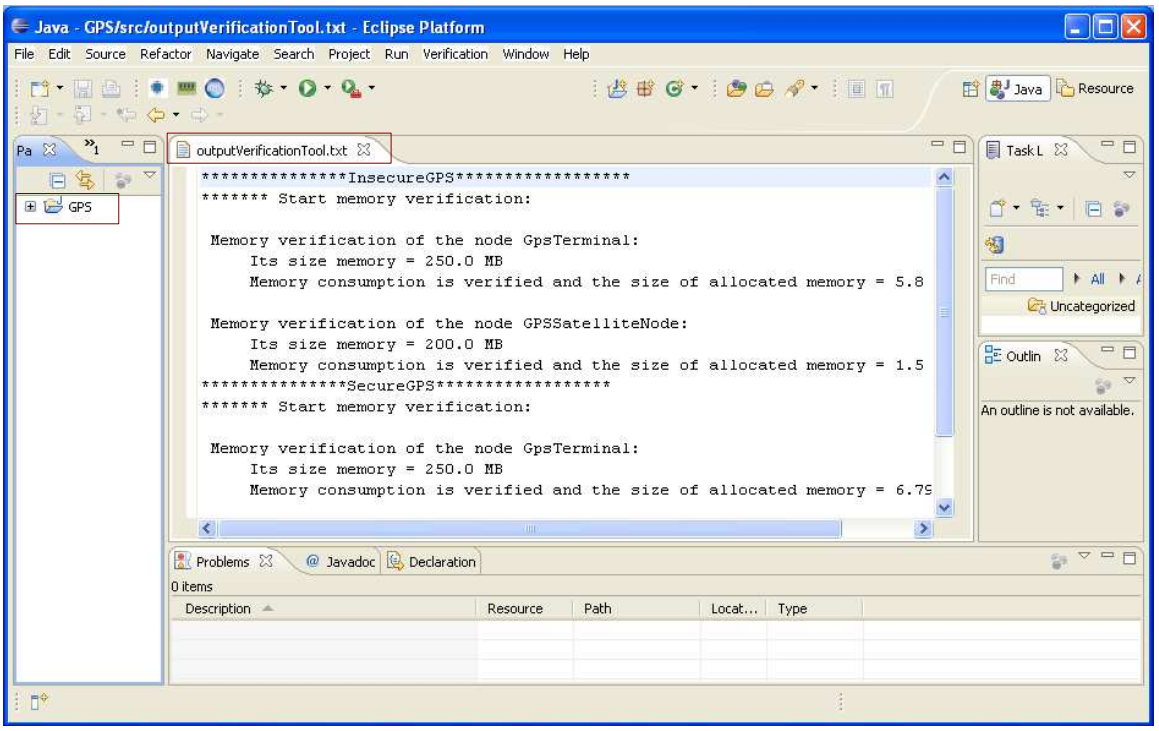

FIGURE 6.7 – Résultats fournis du framework Cheddar

6.5 Développement d'un intergiciel pour les systèmes TR²E

Pour développer notre intergiciel, nous avons adapté et étendu l'intergiciel PolyORB_HI.

6.5.1 Profil Ravenscar

Notre intergiciel est inspiré du profil Ravenscar. Ravenscar recommande l'utilisation des tâches périodiques ou sporadiques. Il interdit l'utilisation des tâches qui se déclenchent aléatoirement. Pour cela, pour l'implémentation de notre intergiciel, nous avons défini trois classes Java nommées *PeriodicTask*, *SporadicTask* et *AperiodicTask* pour la gestion des tâches périodiques, des tâches sporadiques et des tâches apériodiques[7], respectivement.

La communication entre les tâches doit être asynchrone et utilise un ensemble statique d'objets partagés protégés. Cette communication est conforme au protocole de plafonnement de priorité PCP (Priority Ceilling Protocol) [55]. Pour cela, nous

7. Dans notre cas, nous supposons qu'une tâche apériodique a une date d'arrivée

avons défini une classe *GlobalQueue* (des files d'attente partagées) pour implanter la communication entre les tâches de l'intergiciel, comme montré dans la Figure 6.8. Les deux méthodes *storeOut* et *storeIn* permettent respectivement d'envoyer et de recevoir des données.

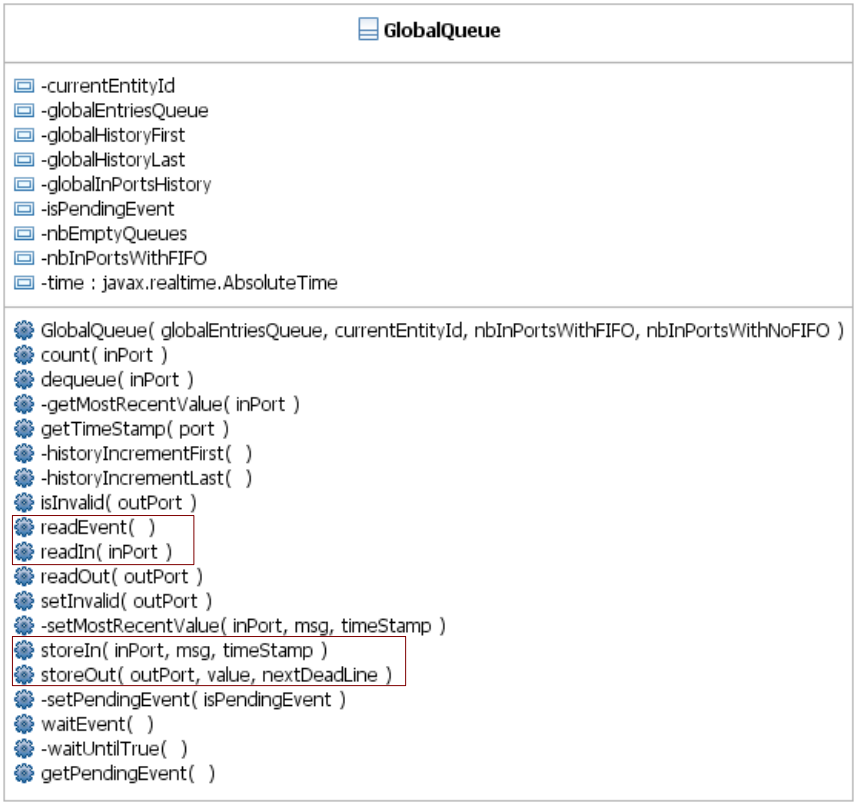

FIGURE 6.8 – Classe *GlobalQueue* de l'intergiciel

6.5.2 Développement de RCES4RTES

L'intergiciel RCES4RTES est une extension de PolyORB_HI pour supporter l'exécution des systèmes TR^{2}E dynamiquement reconfigurables (voir Figure 6.9). En effet, il profite des avantages de PolyORB_HI pour les systèmes TR^{2}E et de sa faible empreinte mémoire. L'implémentation s'inspire du profil Ravenscar et fournit de nouvelles routines pour supporter l'exécution de la reconfiguration dynamique, la cohérence et la supervision des systèmes TR^{2}E dynamiquement reconfigurables. La Figure 6.10 représente les deux classes contenant ces routines. La classe *ReconfDyn* contient les routines assurant la reconfiguration et la cohérence, tandis que la classe *Observer* contient des routines assurant la supervision.

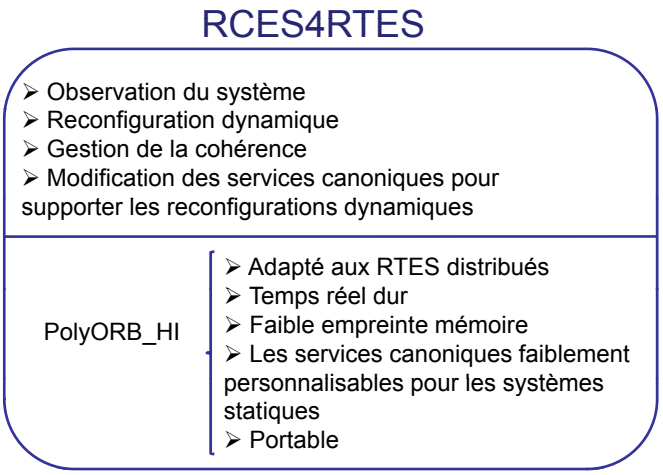

FIGURE 6.9 – Caractéristiques de l'intergiciel RCES4RTES

- L'intergiciel RCES4RTES présente un extension et spécialisation des services canoniques du PolyORB_HI pour les systèmes TR²E dynamiquement reconfigurables :
  - Le service adressage : il ajoute la création et la suppression dynamiques des références des entités,
  - LE service liaison : il ajoute et/ou supprime des connexions par la sélection dynamique des instances des services impliqués dans les communications distantes,
  - Le service activation : il ajoute l'activation des entités créées dynamiquement,
  - Le service transport : il permet la mise à jour dynamique de la liste des ports destinataires (canaux de communication),
  - Le service représentation : il ajoute l'allocation et le calcul dynamique de la taille des tampons de communication,
  - Le service exécution : il permet le blocage et le déblocage du composant à reconfigurer et de ses sources lors de la modification du comportement et/ou de la structure d'une architecture logicielle.
- Il introduit des nouvelles routines pour la reconfiguration :
  - addComponent : elle permet d'ajouter un composant,
  - updateProperties : elle permet de mettre à jour les propriétés d'un composant,
  - addConnection : elle crée une connexion entre deux ports afin de permettre l'envoi des messages entre les deux composants auxquels ils sont associés,

FIGURE 6.10 – Classes *ReconfDyn* et *Observer* de l'intergiciel RCES4RTES

- removeConnection : elle supprime une connexion entre deux ports reliant deux composants,
- connect : elle permet d'établir une connexion entre le nœud courant et un nœud distant,
- disconnect : elle supprime la connexion entre le nœud courant et un nœud distant,

  - changeProperties : elle permet de modifier certaines propriétés pour une tâche donnée,
  - addTask : elle ajoute une nouvelle tâche sur la plate-forme d'exécution,
  - removeTask : elle supprime une tâche qui s'exécute sur un nœud donné,
  - migrateTask : elle permet de migrer une tâche d'un nœud source vers un nœud cible,
  - replaceTask : elle remplace une tâche qui s'exécute sur un nœud donné par une autre tâche.
- Il ajoute des nouvelles routines pour la cohérence :
  - blockTask : elle permet de bloquer une tâche (affectée par le processus de reconfiguration) ainsi que ses sources (les tâches qui lui envoient des requêtes),
  - unblockTask : elle permet de débloquer une tâche (affectée par le processus de reconfiguration) ainsi que ses sources.
- Il introduit aussi des nouvelles routines pour la supervision :
  - numberInstances : elle permet de déterminer le nombre d'instances d'une tâche (d'un type donné) qui s'exécutent sur le nœud courant,
  - numberConnections : elle permet de déterminer le nombre de connexions reliant une tâche donnée avec les autres tâches,
  - observationVariables : retourner les dates récentes d'accès à une variable partagée de l'application.

6.5.3 Services de l'intergiciel RCES4RTES

Un ensemble de routines est fourni par l'intergiciel RCES4RTES pour supporter l'implantation des reconfigurations architecturales et comportementales décrites précédemment. Nous détaillons dans cette partie quelques routines.

Ajout d'un composant

Le Listing 6.1 présente la routine permettant d'ajouter un composant. Cette routine permet d'ajouter une instance d'un composant sur un noeud et de le connecter avec les autres composants. Elle a comme paramètres (ligne 1) : le composant, le noeud où la nouvelle instance sera déployée, la liste des destinations de chaque port de cette nouvelle instance et la liste des ports ayant cette instance dans leurs destinations. Après avoir vérifié que le noeud n supporte ce composant, une nouvelle instance du composant t sera créée et déployée sur le noeud n (lignes 2–4). Les connexions de cette instance avec les autres composants sont aussi ajoutées (lignes

6–15).

```
1  addComponent(t: ComponentType; n:Node; dest[][],
2                                  source[][]:hash table)
3      if (exist(t,n)) then
4          c=newInstance(t);
5          deploy(c,n);
6          For i=1 to dest.length do
7              P1=getOutputPort(dest[i][1]);
8              P2=getInputPort(dest[i][2]);
9              addConnection(P1,P2);
10         EndFor
11         For i=1 to source.length do
12             P1=getOutputPort(source[i][2]);
13             P2=getInputPort(source[i][1]);
14             addConnection(P2,P1);
15         EndFor
16     endIf
```

Listing 6.1 – Ajouter un composant

Mise à jour d'un composant

Le listing 6.2 présente la routine permettant de mettre à jour les propriétés d'un composant. L'ensemble des propriétés à mettre à jour est divisé en deux catégories : les propriétés critiques et les propriétés non critiques (ligne 2).

Les propriétés non critiques peuvent être mises à jour au cours de l'exécution sans bloquer le composant correspondant (lignes 3–5). A l'inverse, les propriétés critiques qui affectent l'exécution du composant seront mises à jour (lignes 8–10) après le blocage de ce composant (ligne 7).

```
1  updateProperties(c: ComponentInstance; P[]:properties)
2      P.classifyProperties(criticalProperties,
3                                  nonCriticalProperties);
4      For i=1 to nonCriticalProperties.length do
5          updateProperty(nonCriticalProperties[i],c);
6      EndFor
7      if (exist(criticalProperties)) then
8          lockComponent(c);
9          For i=1 to criticalProperties.length do
```

```
10              updateProperty(criticalProperties[i],c);
11          EndFor
12          unlockComponent(c);
13      endIf
```

Listing 6.2 – Mettre à jour les propriétés d'un composant

Ajout d'une connexion

Le Listing 6.3 présente la routine permettant d'ajouter une connexion entre deux composants (deux ports). Tout d'abord, cette routine vérifie si les deux composants sont imbriqués et dans ce cas les deux ports *P1* et *P2* doivent être de même type (deux ports d'entrée ou deux ports de sortie) (lignes 2–4). Autrement (les deux composants ne sont pas imbriqués), le port *P1* doit être un port de sortie et le port *P2* doit être un port d'entrée (lignes 5–6). Afin d'éviter la perte de messages entre les composants, le composant source (composant ayant le port *P1*) doit être bloqué avant d'ajouter la connexion (ligne 8). Juste après l'ajout de connexion, le composant source sera débloqué (ligne 10) pour envoyer et recevoir des messages. La connexion sera créée en ajoutant l'identifiant du port *P2* à l'ensemble des destinations du port *P1* (ligne 9) par la mise à jour de la structure de données *destinationPortsDS*.

```
1   addConnection(P1,P2:Port)
2       if ((isNested(P1.getComponent(),P2.getComponent())
3               and ((isOutputPort(P1) and isOutputPort(P2))
4                       or (isInputPort(P1) and isInputPort(P2))))
5                       or
6           (not isNested(P1.getComponent(),P2.getComponent())
7               and isOutputPort(P1) and
8               isInputPort(P2))
9               and
10          not isExistConnection(P1,P2)) then
11          lockComponent(P1.getComponent());
12          P1.addDestinationPort(P2);
13          unlockComponent(P1.getComponent());
14      endIf
```

Listing 6.3 – Ajouter une connexion

6.5.4 Configuration de l'intergiciel RCES4RTES

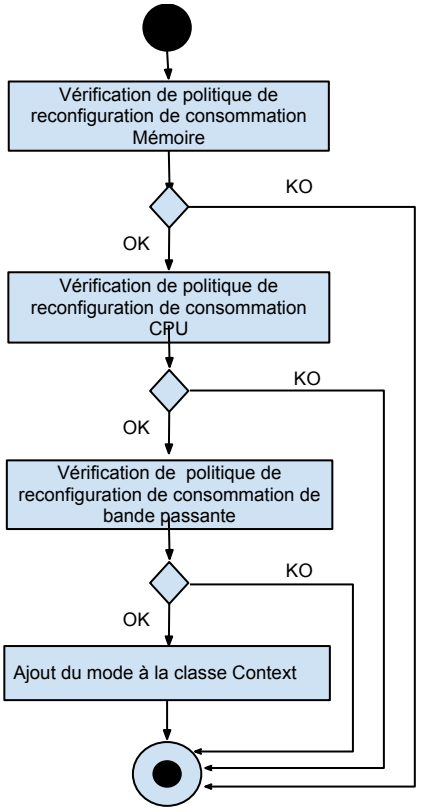

FIGURE 6.11 – Ajout des modes à l'intergiciel RCES4RTES

Dans le but de générer les différents modes du système respectant les politiques de reconfiguration, nous avons développé trois algorithmes (voir la sous-section 5.5.1) permettant de générer les modes respectant les taux de consommation CPU, de consommation mémoire et de consommation de la bande passante spécifiés à un haut niveau d'abstraction. Ces modes générés sont ajoutés à l'intergiciel RCES4RTES directement. Les procédures définies par les Listings 6.4, 6.5 et 6.6 permettent de vérifier si un mode respecte les politiques de reconfiguration correspondant respectivement à la consommation mémoire, la consommation CPU et la consommation de bande passante comme décrit dans la section 5.5.1. L'algorithme principal ajoute les modes respectant à la fois les trois politiques de reconfiguration à la structure de données *modes* dans la classe *Context* de l'intergiciel. Pour chaque mode et comme le montre la Figure 6.11, nous commençons par la vérification de la politique de reconfiguration de consommation mémoire. Si cette politique est satisfaite, nous vérifions par la suite la politique de reconfiguration de la consommation CPU. Si cette deuxième politique est aussi vérifiée avec succès, nous passons à la dernière politique

de la consommation de bande passante. Si cette troisième politique est bien vérifiée, nous ajoutons le mode à la classe *Context*.

```
1  usageMemoryCalcul(Mode mode, Float rate)
2          List allocatedTask
3          for each Node n in the execution platform
4              allocatedTask = getAllocatedTasks(mode, n)
5              for each task t in the list allocatedTask
6                  classifyTasks(t)
7              endFor
8              // if all tasks are periodic and sporadic
9              memApMax = 0
10             if existenceTaskAp
11                 // Compute memory allocated by aperiodic task
12                 memApMax = memoryOfApTask(task)
13             endIf
14             memP = 0
15             if existenceTaskP
16                 memP = sumTaskP(tasksP)
17             endIf
18             memSp = 0
19             if existenceTaskSp
20                 memSp = sumTaskSp(tasksSp)
21             endIf
22             memoryUsage = (memApMax + memSp+ memP)/n.memory
23              //rate is the pourcentage given by user
24             if memoryUsage < rate
25                 memory consomption policy is verified for the node n
26             else
27                 exit()
28             endIf
29         endFor
30         if memory consomption policy is verified for all node then it is verified
             for the mode "mode"
31 endProcedure
```

Listing 6.4 – Vérification de la politique de reconfiguration pour la consommation mémoire

```
usageCpuCalcul(Mode mode, Float rate)
        List allocatedTask
        for each CPU c in the execution platform
            allocatedTask = getAllocatedTasks(mode, cpu)
            for each task t in the list allocatedTask
                classifyTasks(t)
            endFor
            // Compute meta-period of the aperiodic tasks
            cpuUsage = 0
            mp = getPeriod(periods, times)
            for each aperiodic task t of the mode
                cpuUsage = cpuUsage + (t.wcet1 + t.wcet2)/mp
            endFor
            for each periodic task t of the mode
                cpuUsage = cpuUsage + (t.wcet1 + t.wcet2)/t.period
            endFor
            for each sporadic task t of the mode
                cpuUsage = cpuUsage + (t.wcet1 + t.wcet2)/t.period
            endFor
             //rate is the pourcentage given by user
            if cpuUsage < rate
                CPU consomption policy is verified for the Cpu c
            else
                exit()
            endIf
        endFor
        if CPU consomption policy is verified for all Cpu then it is verified for
          the mode "mode"
endProcedure
```

Listing 6.5 – Vérification de la politique de reconfiguration pour la consommation CPU

```
usageBwCalcul(Mode mode, Float rate, Float BwBus)
        List cnx
        for each Bus b in the execution platform
            cnx = getAllocatedConnection(mode,b)
            for each connection c in the list cnx
```

```
65                  bwUsage = bwUsage + c.bandwith
66              endFor
67              bwUsage= bwUsage/BwBus //rate is the pourcentage given by user
68              if bwUsage < rate
69                  bandwith Consomption policy is verified for the bus b
70              else
71                  exit()
72              endIf
73          endFor
74          if bandwidth Consomption policy is verified for all bus then it is verified
              for the mode "mode"
75  endProcedure
```

Listing 6.6 – Vérification de la politique de reconfiguration pour la consommation de la bande passante

6.6 Outils de génération

Pour une génération optimale du code, nous avons réalisé un autre plugin Eclipse. Ce dernier dépend des plugins ATL et Acceleo afin de faire respectivement une transformation d'un modèle à un autre et une génération du code à partir d'un modèle. En effet, il permet de faire deux types de transformation comme le montre la Figure 6.12 :

- Transformation M2M : permettant de transformer des modèles RCA4RTES (conformes au méta-modèle RCA4RTES) vers des modèles d'implantation (conformes au méta-modèle d'implantation).
- Transformation M2T : permettant de générer du texte (code Java) à partir des modèles d'implantation.

Afin de générer le code, les modèles RCA4RTES spécifiés en utilisant le langage UML, le profil RCA4RTES et le profil MARTE seront transformés vers des modèles d'implantation présentant le code généré. Pour définir ces modèles, nous avons aussi utilisé le langage UML et nous avons défini un nouveau profil UML qui étend le méta-modèle d'implantation proposé dans la section 5.5. Les modèles RCA4RTES sont transformés automatiquement vers des modèles d'implantation par des règles de transformation. Dans ce qui suit, nous présentons le profil d'implantation proposé, les règles de transformation permettant d'obtenir automatiquement des modèles

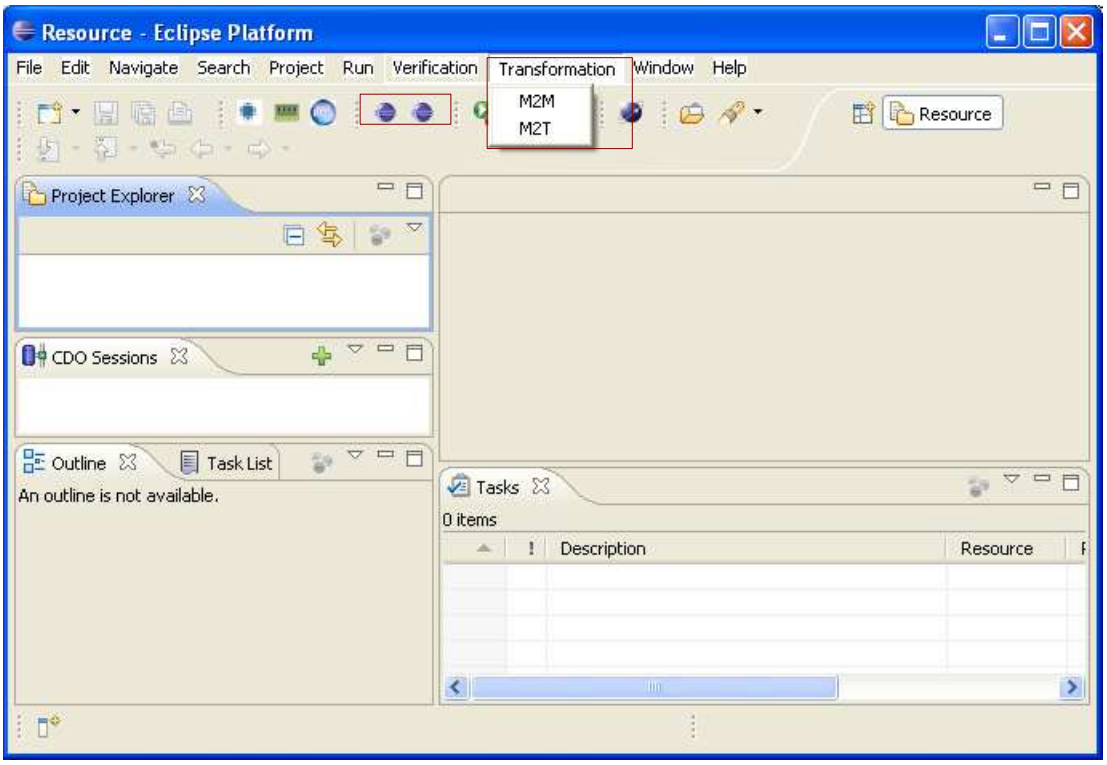

FIGURE 6.12 – Capture d'écran du plug-in de génération de code

d'implantation à partir des modèles RCA4RTES et la génération du code à partir des modèles d'implantation.

6.6.1 Profil d'implantation

Nous proposons un profil UML dérivé du notre méta-modèle d'implantation (Figure 6.13).

Le stéréotype *System*

- **Description :** ce stéréotype décrit l'implantation complète du système. Il représente le paquetage principal du système qui contient un ensemble de processus.
- **Méta-classe étendue :** *Package*.

Le stéréotype *Process*

- **Description :** ce stéréotype permet de décrire les différents processus d'un système qui seront déployés sur des nœuds différents. Chaque processus représente un ensemble de tâches qui peuvent être périodiques, sporadiques et/ou apériodiques.
- **Méta-classe étendue :** *Package*.

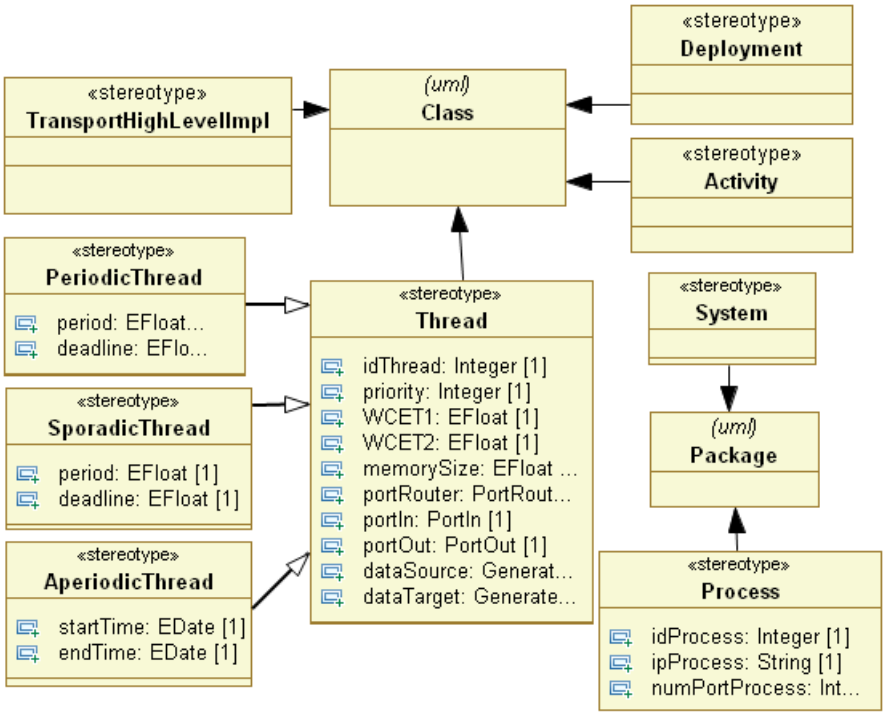

FIGURE 6.13 – Description du profil d'implantation

Le stéréotype *Thread*

- **Description :** ce stéréotype permet de spécifier les tâches (threads) du système.
- **Méta-classe étendue :** *Class*.
- **Propriétés :**
  - **id :** elle définit l'identifiant de la tâche,
  - **priority :** elle définit la priorité de la tâche,
  - **WCET1 :** elle définit le pire temps d'exécution de la tâche sur le processeur où elle a été allouée,
  - **WCET2 :** elle définit le pire temps d'exécution qui dépend d'autres périphériques que le processeur, comme les bus ou les mémoires,
  - **memorySize :** elle définit l'empreinte mémoire de la tâche,
  - **portIn :** elle détermine les ports d'entrée de la tâche,
  - **portOut :** elle détermine les ports de sortie de la tâche.

Le stéréotype *PeriodicThread*

- **Description :** ce stéréotype permet de spécifier les tâches périodiques.
- **Méta-classe étendue :** *Class*.
- **Propriétés :**

  - **period :** elle détermine la période de la tâche,
  - **deadline :** elle détermine l'échéance de la tâche.
- **Généralisation :** le stéréotype *Thread*.

Le stéréotype *SporadicThread*

- **Description :** ce stéréotype permet de spécifier les tâches sporadiques.
- **Méta-classe étendue :** *Class*.
- **Propriétés :**
  - **period :** elle détermine la durée minimale entre deux événements successives déclenchant la tâche,
  - **deadline :** elle détermine l'échéance de la tâche.
- **Généralisation :** *Thread*.

Le stéréotype *AperiodicThread*

- **Description :** ce stéréotype permet de spécifier les tâches apériodiques.
- **Méta-classe étendue :** *Class*.
- **Propriétés :**
  - **starTime :** elle détermine le temps d'arrivée et de lancement d'exécution d'une tâche apériodique.
  - **endTime :** elle détermine le temps de fin d'une tâche apériodique.
- **Généralisation :** *Thread*.

Le stéréotype *Deployment*

- **Description :** ce stéréotype permet de spécifier l'architecture du système. Il permet de définir le mode initial du système.
- **Méta-classe étendue :** *Class*.

Le stéréotype *Activity*

- **Description :** ce stéréotype permet de spécifier les reconfigurations dynamiques.
- **Méta-classe étendue :** *Class*.

Le stéréotype *TransportHighLevelImpl*

- **Description :** ce stéréotype permet de spécifier les communications entre les tâches (envoi de données pour chaque tâche émettrice et réception des données pour chaque tâche réceptrice).
- **Méta-classe étendue :** *Class*.

6.6.2 Règles de transformation

Les modèles RCA4RTES spécifiés durant la phase de modélisation sont raffinés vers des modèles d'implantation à travers des transformations décrites en ATL. Le tableau 6.1 résume les correspondances entre le méta-modèle RCA4RTES et le méta-modèle d'implantation afin de déduire les transformations. Par exemple, un élément d'un modèle de type *SoftwareSystem* sera transformé vers un élément dans le nouveau modèle de type *System*. En d'autres termes, une machine à états (UML StateMachine) stéréotypée par *SoftwareSystem* sera transformée en un paquetage (UML Package) stéréotypé par *System*. Dans le listing 6.7, nous montrons la règle correspondante à cette transformation.

```
1  rule SoftSys2System {
2  from s : UML!StateMachine
3  (s.hasStereotype('SoftwareSystem'))
4  to t :
5  UML!Package (name <- s.name)
6  do {
7      -- Add the created package in the package of the model
8      thisModule.package.packagedElement <- t;
9      -- Apply the stereotype System on the generated package
10     t.applyStereotype(thisModule.getStereotype('System'));
11         }
12     }
```

Listing 6.7 – Règle ATL de transformation du modèle *SoftwareSystem* vers le modèle *System*

6.6.3 Génération de code

A partir des modèles d'implantation, nous utilisons de nouvelles transformations (templates) pour générer le code du système, pour l'intergiciel RCES4RTES, en adoptant Acceleo. De ce fait, nous générons du code Java tout en réutilisant les routines de l'intergiciel RCES4RTES.

Le Listing 6.8 représente le template *generate*. C'est le template principal qui permet la génération de code. Pour cela, il utilise d'autres templates permettant de récupérer les informations nécessaires à partir des modèles d'implantation. Parmi ces templates utilisés, nous citons le template *classBody* pour la génération des classes Java avec leurs attributs et leurs méthodes, ainsi que le template *common* permettant la génération des types de retour, des types des paramètres des méthodes ainsi que des commentaires. Nous présentons dans le tableau 6.2 la correspondance

Méta-modèle RCA4RTES	Méta-modèle d'implantation
SoftwareSystem	*System*
ExecutionPlatform	*Process*
StructuredComponent - nature=periodic - nature=sporadic - nature=aperiodic	 *PeriodicThread* *SporadicThread* *AperiodicThread*
Allocation	Ajouter une association *Allocated* entre les deux méta-classes *Thread* et *Processor*
Connector	*TransportHighLevelImpl* - send() - deliver()
ConfigurationDeploymentPlan	*Deployment*
ModeTransition	*Activity*
FlowPort - direction=in - direction=out - direction=in/out	 - ajout du port à la liste *portIn* du Thread Correspondant - ajout du port à la liste *portOut* du Thread Correspondant - ajout du port à la liste *portIn* et à la liste *portOut* du Thread Correspondant
ClientServerPort - kind= required - kind=provider - kind=required/provider	 - ajout du port à la liste *portIn* du Thread Correspondant - ajout du port à la liste *portOut* du Thread Correspondant - ajout du port à la liste *portIn* et à la liste *portOut* du Thread Correspondant

TABLE 6.1 – Correspondance entre le méta-modèle RCA4RTES et le méta-modèle d'implantation

entre les modèles d'implantation et le code Java afin d'écrire les templates Acceleo.

```
1 generate('http://www.eclipse.org/uml2/3.0.0/UML')/] [importcommon/]
2 [import classBody/] [template public generateClass(c : Class)]
3 [comment @main /] [file (c.getFullPathFile().trim(),false)]
4 [_commentFileBlock()/] [c.packageBlock()/] [c.importBlock()/]
5 [_commentBodyBlock()/] [c.generateClassBody()/] [/file] [/template]
```

Listing 6.8 – Template principal *generate*

Méta-modèle d'implantation	**Java**
UML Package stéréotypé *System*	Un paquetage Java
UML Package stéréotypé *Process*	Un sous paquetage Java du paquetage System
UML Class stéréotypée *PeriodicThread*	Une classe java héritant de la classe *PeriodicTask* de l'intergiciel. Les valeurs marquées du stéréotype *PeriodicThread* seront ajoutées comme des propriétés à la classe java générée.
UML Class stéréotypée *SporadicThread*	Une classe java héritant de la classe *SporadicTask* de l'intergiciel. Les valeurs marquées du stéréotype *SporadicThread* seront ajoutées comme des propriétés à la classe java générée.
UML Class stéréotypée *AperiodicThread*	Une classe java héritant de la classe *AperiodicTask* de l'intergiciel. Les valeurs marquées du stéréotype *AperiodicThread* seront ajoutées comme des propriétés à la classe java générée.
UML Class stéréotypée *Deployment*	Une classe java nommée *Deployment* contenant une méthode *initialOperationalMode()* permettant de créer l'instance (configuration) initiale du système.
UML Class stéréotypée *TransportHighLevelImpl*	Une classe java nommée *TransportHighLevelImpl* contenant une méthode *send()* pour chaque tâche émettrice et une méthode *deliver()* pour chaque tâche réceptrice.
UML Class stéréotypée *Activity*	Une classe java nommée *Activity* contenant une méthode *reconfiguration()* pour activer la tâche *ReconfigurationTrigger* de l'intergiciel afin d'appliquer les actions de reconfiguation.

TABLE 6.2 – Correspondance entre les modèles d'implantation et le code Java

6.7 Conclusion

Dans ce chapitre, nous avons présenté la suite d'outils que nous proposons pour mettre en œuvre le processus de développement des systèmes TR²E dynamiquement reconfigurables. Il s'agit d'une chaîne d'outils IDM permettant de modéliser la reconfiguration et ses politiques, de modéliser les architectures logicielles/matérielles, de modéliser l'allocation, de vérifier les propriétés non-fonctionnelles, de configurer la plate-forme d'exécution et enfin d'implanter le code du système. Les briques de la chaîne sont conformes aux langages de modélisation et au framework de vérification précédemment présentés.

Nous avons développé un nouvel intergiciel dédié pour des systèmes TR²E dynamiquement reconfigurables en étendant l'intergiciel PolyORB_HI par de nouvelles routines pour la reconfiguration dynamique, pour la cohérence, ainsi que pour la su-

pervision de ces systèmes. La chaîne d'outils inclut également des générateurs aidant à l'automatisation des constructions et permettant de générer les modes du système, pour configurer l'intergiciel, et le code après la transformation des modèles de haut niveau vers des modèles proches de l'implantation du système.

Dans le chapitre suivant, nous allons illustrer le processus de développement et la suite d'outils en considérant l'étude de cas : MyGPS (une version simplifiée du système de localisation mondial [8]).

8. Global Positioning System

Chapitre 7

Étude de cas : Système de localisation (MyGPS)

7.1 Introduction

L'objectif de ce chapitre est d'illustrer l'approche proposée dans les chapitres précédents sur un cas d'étude. Pour ce faire, nous montrons comment utiliser cette approche pour modéliser un exemple de système de localisation (MyGPS) à partir d'un cahier de charges pour produire un modèle prêt pour la vérification. Ensuite, nous appliquons l'approche en utilisant la suite d'outils associée pour une construction automatique des systèmes TR^{2}E dynamiquement reconfigurables, notamment la génération des modes pour configurer la plate-forme d'exécution et la génération de code pour automatiser partiellement l'implantation du système.

Sur ce constat, nous présentons tout d'abord un processus de modélisation itératif pour mettre en avant l'apport de la vérification vis-à-vis des caractéristiques non-fonctionnelles et structurelles des composants du système. Nous détaillons ensuite une implantation et une configuration de l'intergiciel RCES4RTES suivies d'une implantation et d'une exécution du système GPS sur cet intergiciel.

7.2 Système de localisation (MyGPS)

Le système de positionnement mondial GPS (Global Positioning System) [52],que nous présentons dans la Figure 7.1, est un système de géolocalisation permettant de positionner un objet sur un plan ou une carte à l'aide de ses coordonnées géographiques. Ainsi, il aide à déterminer la route à suivre d'un emplacement donné vers

des destinations spécifiques en utilisant des informations fournies, via des signaux radio, par les satellites associés. Dans le GPS, *le satellite* envoie au *terminal* un signal crypté contenant divers renseignements utiles pour la localisation et la synchronisation. Le *terminal* collecte et convertit les signaux radio reçus en informations sur la position, la vitesse et le temps. *La base de contrôle*, quant-à-elle, gère la synchronisation des horloges des satellites par rapport au temps atomique international.

Dans notre illustration, nous allons considérer un cas d'étude (MyGPS) qui est une version simplifiée de ce système. Pour des raisons de clarté, plusieurs fonctionnalités de ce système (GPS) ont été omises. De ce fait, nous définissons seulement les deux cas d'utilisation suivants :

- MyGPS non sécurisé : il représente une utilisation traditionnelle (publique) de MyGPS,
- MyGPS sécurisé : il représente une utilisation restrictive de MyGPS avec des exigences de sécurité.

Le passage d'une utilisation "MyGPS sécurisé" vers une utilisation "MyGPS non sécurisé" et vice versa, se produit par commande à partir du terminal.

FIGURE 7.1 – Fonctionnement du système GPS

7.3 Modélisation du système de localisation

Dans cette section, nous présentons les différentes étapes de modélisation (comme décrit dans le processus de développement) de l'étude de cas MyGPS. nous commençons par la spécification des reconfigurations dynamiques. Ensuite nous présentons la spécification de l'architecture logicielle et de l'architecture matérielle. Enfin, nous montrons l'allocation de la partie logicielle sur la partie matérielle.

7.3.1 Reconfiguration

Selon le cahier de charges du système MyGPS, nous définissons deux *MetaModes* :

- Le MetaMode non sécurisé nommé *Insecure GPS MetaMode* pour représenter les états de MyGPS avec un fonctionnement non sécurisé,
- Le MetaMode sécurisé nommé *Secure GPS MetaMode* pour représenter les états de MyGPS avec un fonctionnement sécurisé.

Pour la première étape de la modélisation, nous utilisons le diagramme d'états transitions d'UML via l'éditeur UML Papyrus étendu par le profil RCA4RTES pour spécifier la reconfiguration (Figure 7.2) .

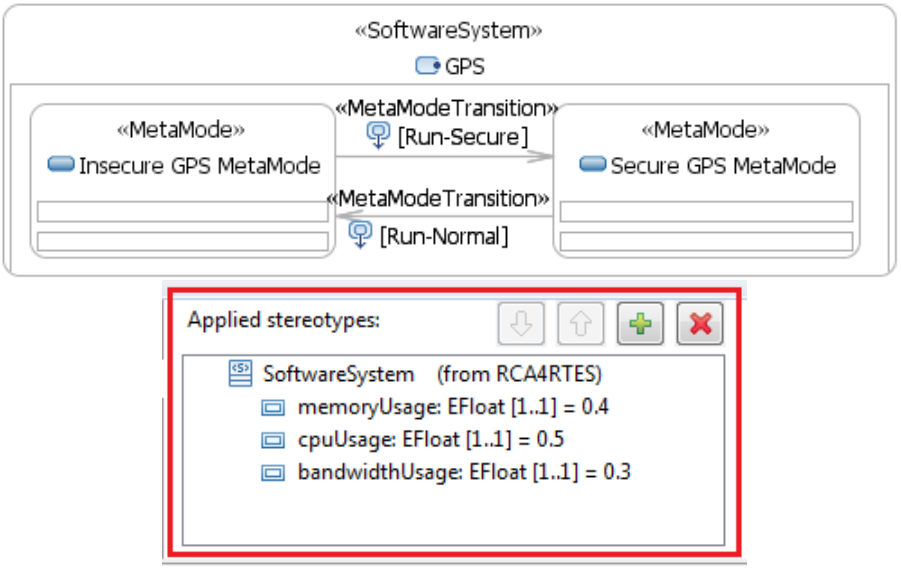

FIGURE 7.2 – Machine à état du Système MyGPS

Ainsi, un MetaMode est modélisé par un état annoté par le stéréotype *MetaMode* et le passage d'un MetaMode vers un autre est modélisé par une transition annotée par le stéréotype *MetaModeTransition*. Dans notre exemple, nous définissons deux MetaModes et deux reconfigurations. La seconde activité de cette première étape

est la spécification des politiques de reconfiguration. Dans notre cas, le modèle est annoté par le stéréotype *SoftwareSystem* et ses valeurs marquées :

- "*memoryUsage*=0.4" : elle est positionnée à 40% (cette valeur définit le taux de consommation de chaque mémoire à ne pas dépasser)
- "*cpuUsage*=0.5" : elle est positionnée à 50% (cette valeur définit le taux de consommation de chaque CPU à ne pas dépasser),
- "*bandwidthUsage*=0.3" : elle est positionnée à 30% (cette valeur définit le taux de consommation de la bande passante de chaque bus à ne pas dépasser).

7.3.2 Modèle du système

Dans la seconde étape, nous spécifions le modèle de l'architecture logicielle et les composants. Ainsi, nous spécifions l'architecture logicielle de chaque *MetaMode* en décrivant ses composants structurés, ses connexions et ses contraintes structurelles et non-fonctionnelles. Nous allons détailler la partie terminal de l'architecture du GPS. Le satellite et la base de contrôle seront représentés par deux composants basiques, *GpsSatellite* et *GpsControlBase*, respectivement.

Insecure MyGPS MetaMode

L'architecture du *MetaMode* non sécurisé (figure 7.3) est définie par sept composants :

- Le composant *GpsControlBase* pour synchroniser l'horloge du satellite.
- Le composant *GpsSatellite* pour envoyer des signaux radio (analogiques) au terminal.
- Le composant *Position* pour recevoir le signal du satellite.
- Le composant *Receiver* pour convertir le signal analogique en signal numérique.
- Le composant *Decoder* pour décoder les informations numériques et séparer les informations du calcul de distance des informations du temps.
- Le composant *TreatmentUnit* pour calculer la distance du satellite et déterminer la position.
- Le composant *Encoder* pour encoder les informations de temps et de position.

Chaque composant structuré est modélisé par un élément stéréotypé *StructuredComponent* (partie centrale de la figure 7.3) et ses propriétés non-fonctionnelles par les valeurs marquées (partie basse de la figure 7.3). Les valeurs des propriétés non-fonctionnelles associées à chaque composant sont définies dans le tableau 7.1.

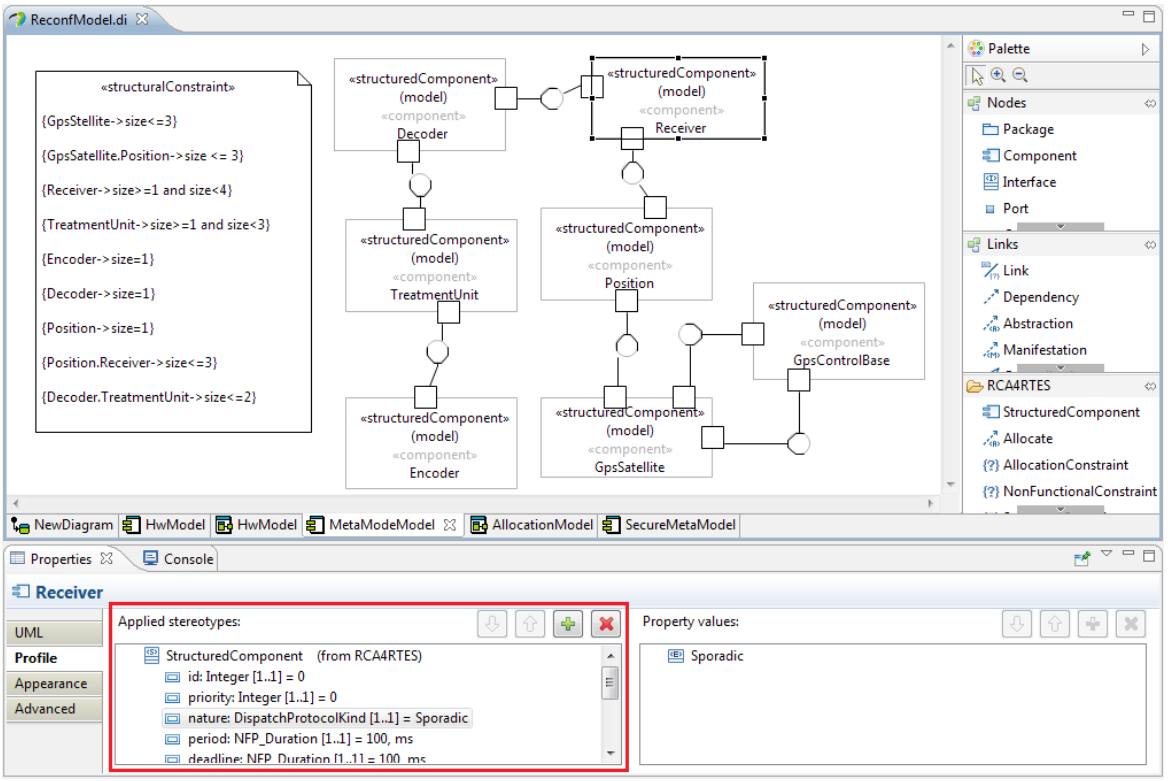

FIGURE 7.3 – MetaMode non sécurisé - Éditeur RCES4RTES

L'architecture du *MetaMode* non sécurisé (figure 7.3) est aussi définie par un ensemble de connexions. Dans cet exemple, les connexions entre les différents composants sont modélisées par un élément stéréotypé *Connector*.

En outre, pour cette architecture (*MetaMode* non sécurisé) nous spécifions un ensemble de contraintes structurelles (partie gauche de la figure 7.3). Par exemple, un mode conforme au *MetaMode* non sécurisé peut avoir entre une et trois instances du composant structuré *Receiver*, mais une seule instance du composant structuré *Position*. Une instance de ce dernier ne peut être connectée qu'au plus trois instances du composant *Receiver*.

Secure MyGPS MetaMode

L'architecture du système pour le *MetaMode* sécurisé (figure 7.4) est composée de huit composants. En comparaison avec les types de composants utilisés par le *MetaMode* non sécurisé, nous avons remplacé le composant *Position* par le composant *SecurePosition* et nous avons ajouté le composant *AccessController* afin d'assurer la réception sécurisée des signaux radio.

Structured Component	Nature	Period Deadline	WCET1 Processor 1Ghz	WCET2	memory size
Receiver	sporadic	100 ms	50 ms	2 ms	4 MB
Position	sporadic	100 ms	20 ms	2 ms	0.5 MB
TreatmentUnit	sporadic	100 ms	20 ms	4 ms	0.75 MB
Decoder	sporadic	100 ms	50 ms	2 ms	0.1 MB
Encoder	sporadic	100 ms	20 ms	2 ms	0.5 MB
GpsSatellite	periodic	400 ms	30 ms	0	0.9 MB
GpsControlBase	periodic	400 ms	30 ms	0	0.9 MB

TABLE 7.1 – Propriétés non-fonctionnelles des composants structurés du *MetaMode* non sécurisé

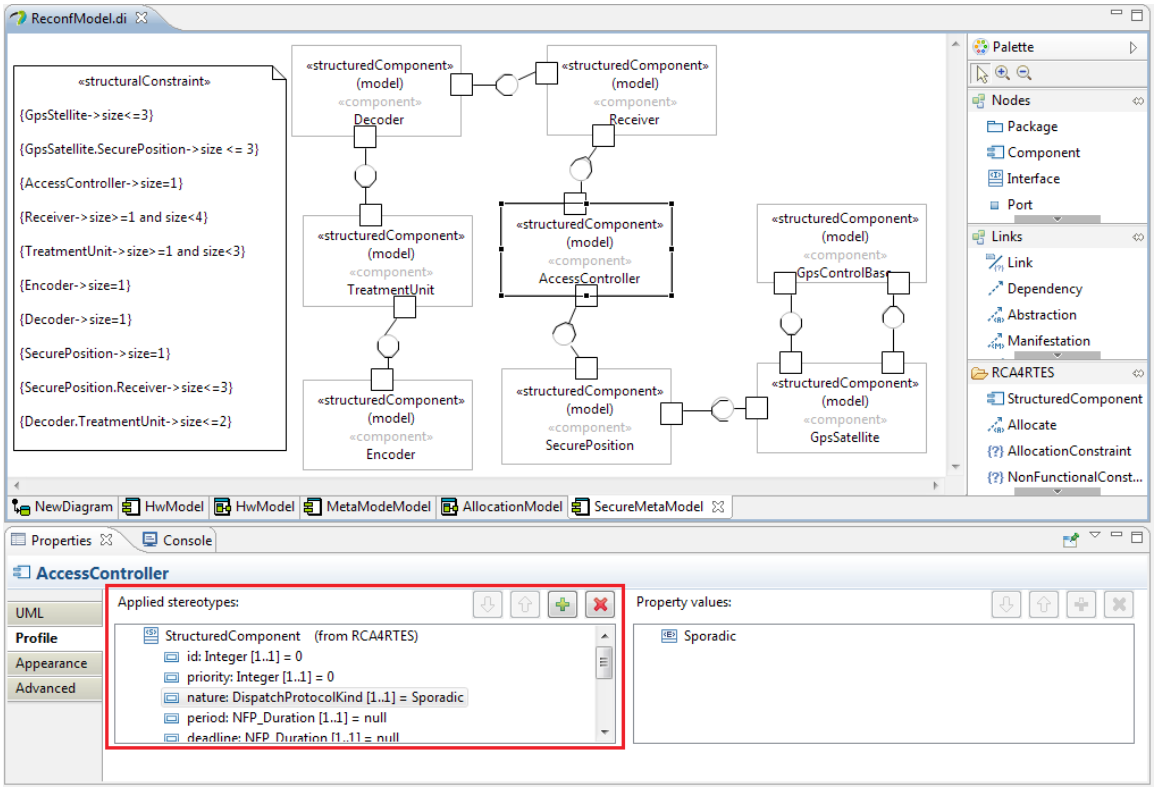

FIGURE 7.4 – MetaMode sécurisé - Éditeur RCES4RTES

7.3.3 Modèle de l'architecture matérielle

La troisième étape consiste à spécifier une instance de l'architecture matérielle. Dans notre exemple, l'architecture matérielle du système MyGSP est composée de trois nœuds : *Terminal*, *Satellite* et *ControlBase*. En utilisant le sous profil HRM du profil MARTE, nous spécifions les ressources matérielles de chaque nœud en termes de processeurs, mémoires, bus, etc. En utilisant les valeurs marquées des stéréotypes du HRM, nous spécifions également les caractéristiques de chacune de ces ressources.

Dans notre exemple, les deux nœuds *Satellite* et *ControBase* sont composés chacun d'une mémoire RAM de taille 10 MB et d'un processeur d'une fréquence de 800 MHz. Le nœud Terminal (figure 7.5) est composé d'une RAM de 10 MB et de deux processeurs ayant chacun une fréquence de 800 MHz. Les bus sont caractérisés par une bande passante égale à 200 Mb/s.

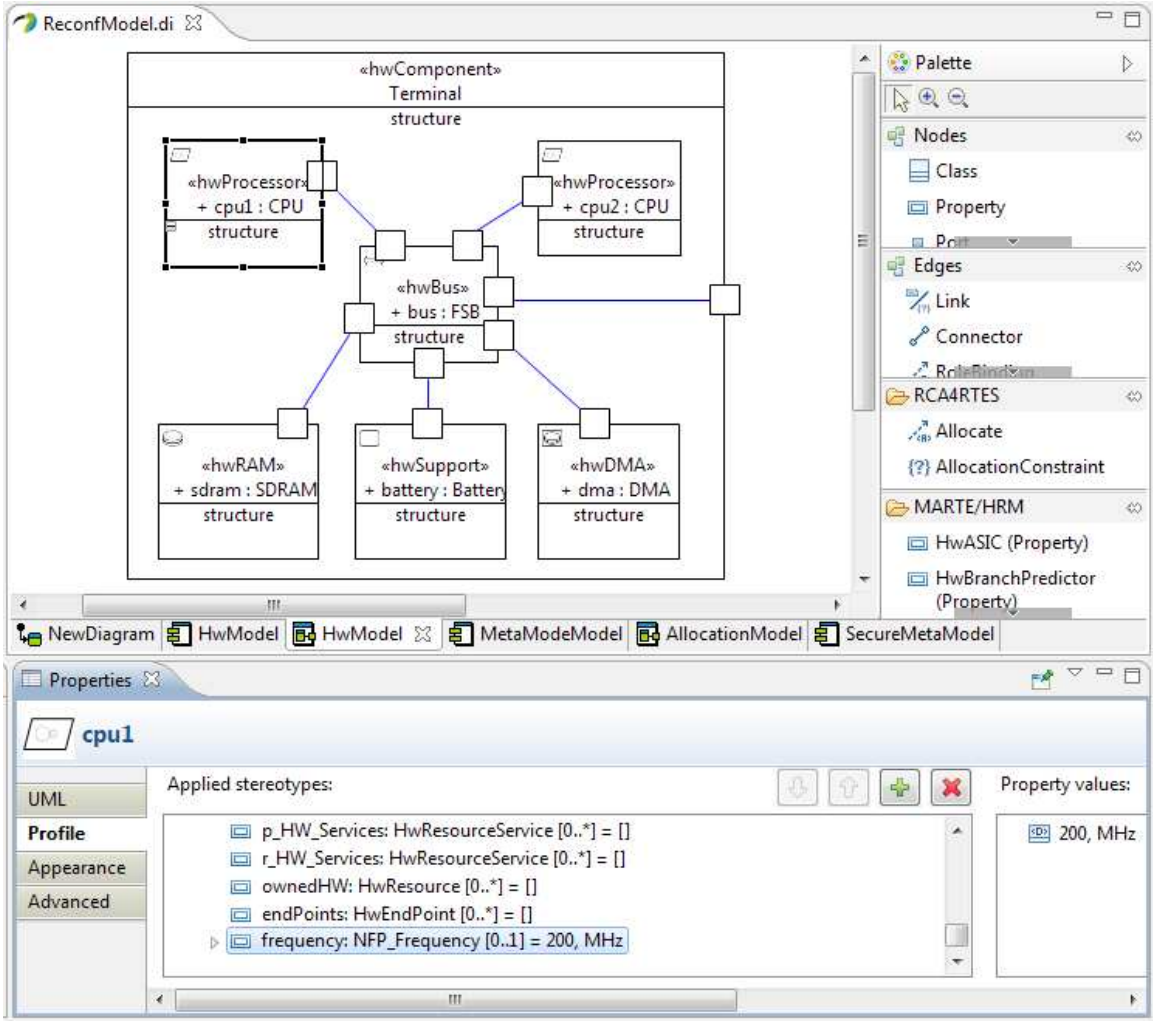

FIGURE 7.5 – Architecture matérielle du nœud *Terminal*

7.3.4 Allocation

La quatrième étape consiste à spécifier l'allocation de chaque architecture (*MetaMode*) sur une instance de l'architecture matérielle. A ce stade, il est également possible d'introduire des contraintes d'allocation. Dans notre exemple, nous allons spécifier l'allocation de l'architecture du MetaMode non sécurisé (figure 7.6) sur une

architecture matérielle composée de deux nœuds : *GpsTerminal* et *GpsSatellite*.

Toutes les instances du composant *Decoder* sont allouées au processeur *cpu1*. Les instances des deux composants *Encoder* et *TreatmentUnit* sont allouées au processeur *cpu2*. Les allocations des instances du composant structuré *Receiver* seront réparties entre les deux processeurs *cpu1* et *cpu2*. Les allocations des instances du composant *Position* sont réparties sur les deux processeurs *cpu1* et *cpu2* du terminal, de telle sorte qu'un tiers des allocations est sur le processeur *cpu1* et les deux autres tiers sont sur le processeur *cpu2*.

7.4 Phase de vérification

La vérification au niveau modèle fonctionne en étroite collaboration avec le framework de modélisation. Nous rappelons que la phase de vérification permet de valider le modèle obtenu lors de la phase précédente, à savoir l'architecture logicielle des MetaModes ainsi que l'allocation vis-à-vis des propriétés non-fonctionnelles de l'architecture logicielle (composants et connexions). Dans le cas où la vérification échoue, le concepteur est invité à réviser ses modèles.

Nous avons effectué un ensemble de simulations en utilisant le plugin *Cheddar+* pour illustrer, d'une part son fonctionnement, et d'autre part la manière dont la vérification aide à éditer des modèles "corrects". Dans notre exemple, nous avons étudié le *MetaMode* non sécurisé et appliqué un ensemble de scénarios. Dans ce qui suit, nous présentons deux scénarios.

7.4.1 Scénario 1 : Vérification de la consommation mémoire

Pour la consommation mémoire, le résultat du test indique que cette propriété n'est pas satisfaite dans le nœud *Terminal* (figure 7.7). Ce résultat est engendré par la grande empreinte mémoire du composant structuré *Receiver*. Il faut donc rectifier la valeur de cette propriété. Après modification (4 MB —>0.9 MB), et re-exécution de la simulation, le test est positif (figure 7.8). Ce résultat nous permet d'affirmer que tous les modes du *MetaMode* non sécurisé vont respecter cette propriété.

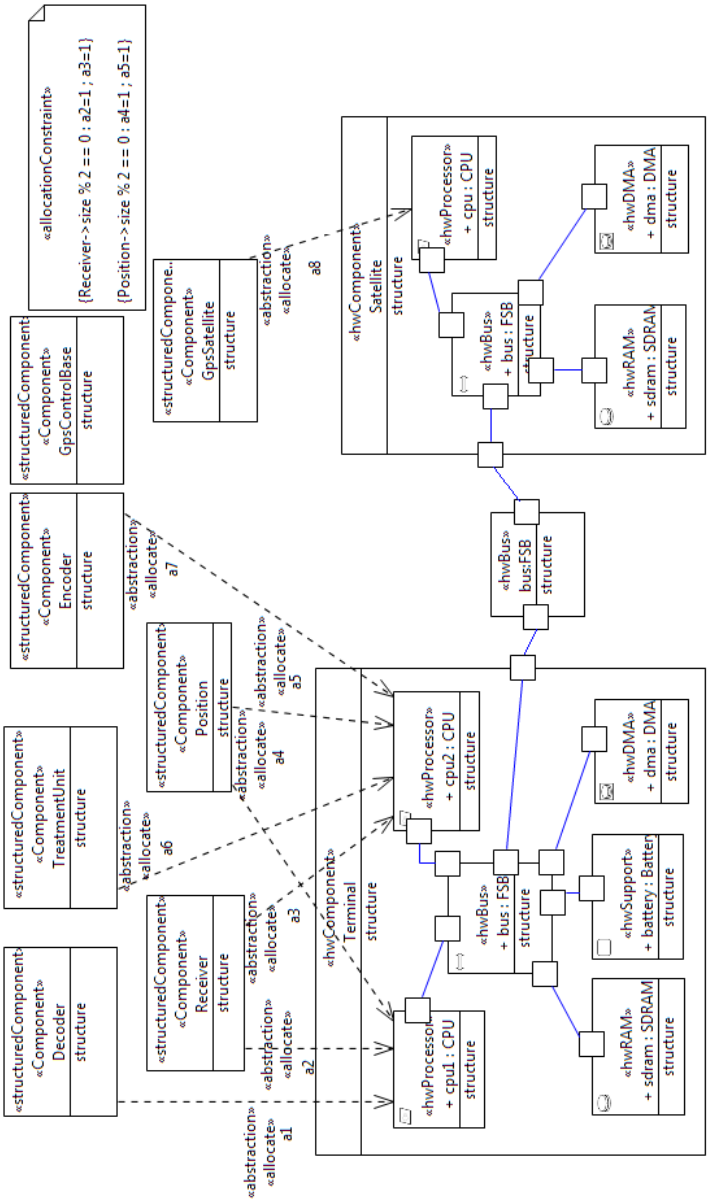

FIGURE 7.6 – Allocation du *MetaMode* non sécurisé à l'architecture matérielle du terminal et du satellite

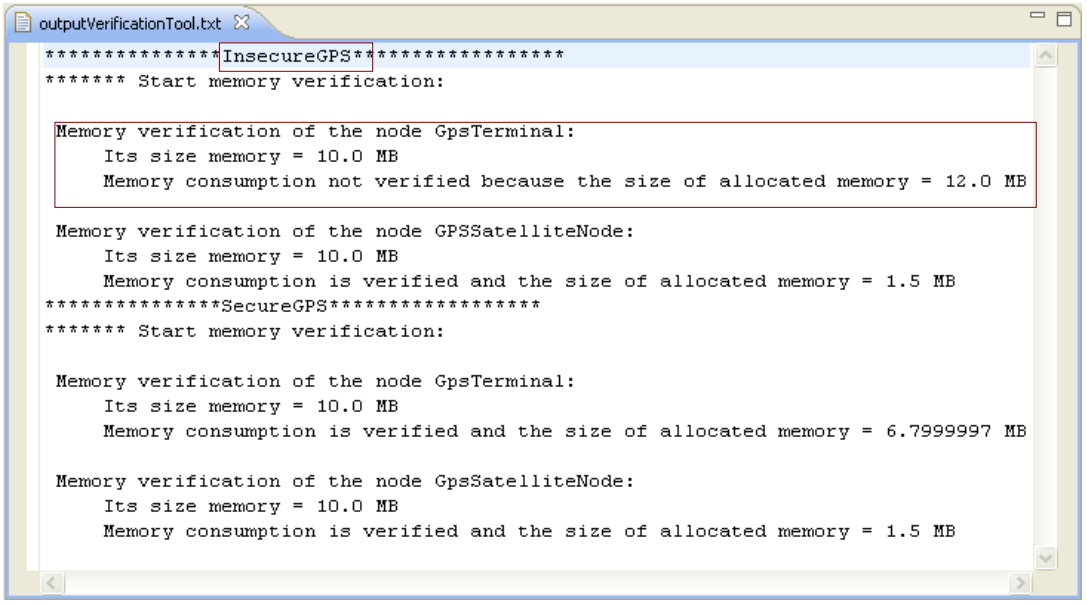

FIGURE 7.7 – Non respect de la consommation mémoire pour le *MetaMode* non sécurisé

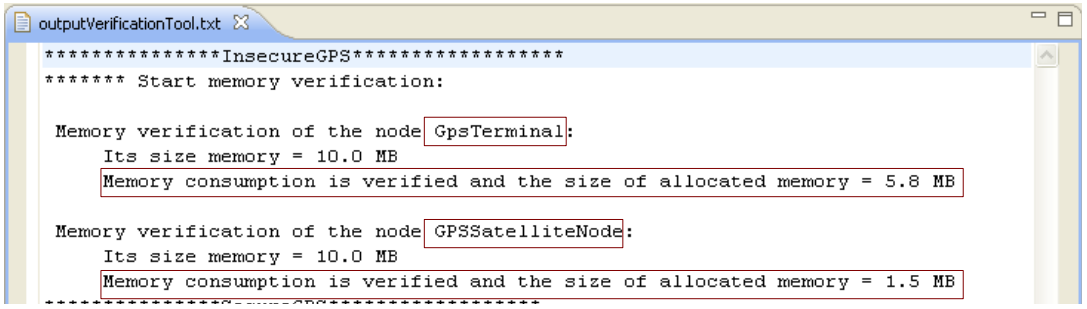

FIGURE 7.8 – Respect de la consommation mémoire pour le *MetaMode* non sécurisé

7.4.2 Scénario 2 : Vérification de la consommation CPU et du respect des échéances

Les résultats du test Cheddar+ présentés respectivement dans les figures 7.9 et 7.10 montrent que la consommation CPU et le respect des échéances des tâches ne sont pas satisfaites pour le processeur *cpu1*. En effet, le facteur d'utilisation de *cpu1* est supérieur à 1 (1.26) et la tâche correspondante à l'instance *decoder1* du composant structuré *Decoder* ne respecte pas son échéance. L'échéance attendue est égale à 100 ms, alors que le test indique une fin d'exécution à 294 ms.

Après modification de la WCET des deux composants *Receiver* et *Decoder* alloués sur le processeur *cpu1* à 20 ms, nous observons des résultats de test positifs

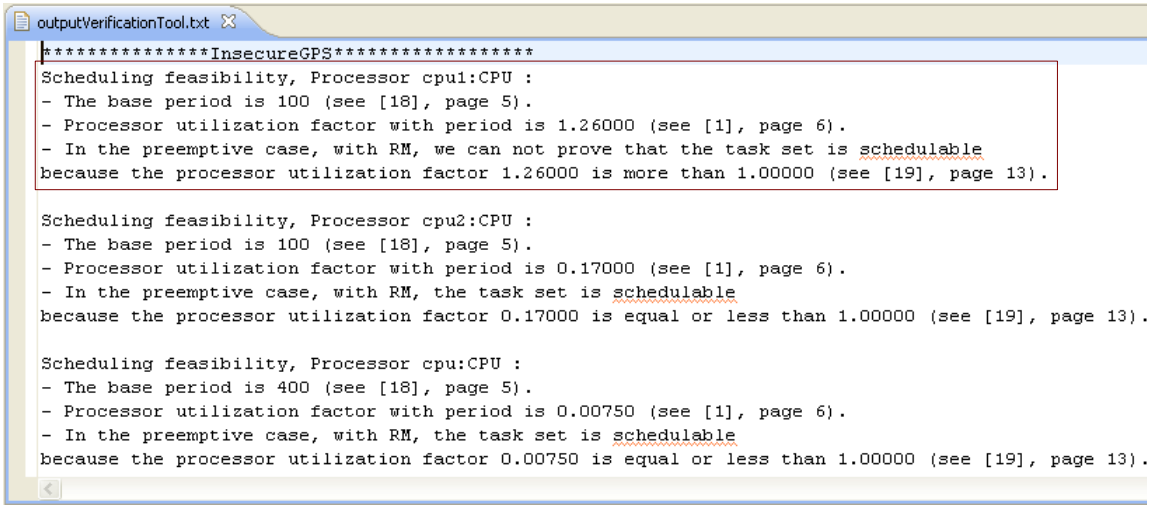

FIGURE 7.9 – Non respect de la consommation CPU pour le *MetaMode* non sécurisé

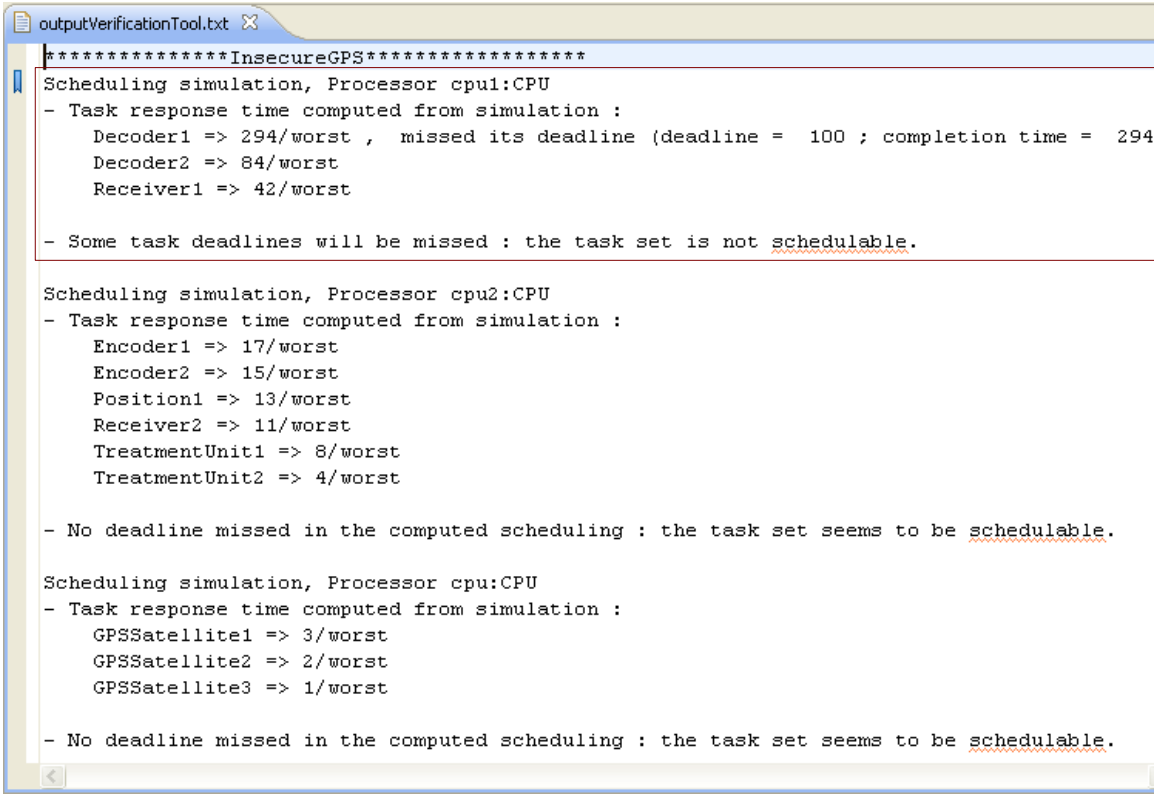

FIGURE 7.10 – Non respect des échéances pour le *MetaMode* non sécurisé

(figure 7.11 et figure 7.12). Le test indique le respect de la consommation CPU et le respect des échéances des tâches. Ce résultat nous permet d'affirmer que tous les modes du *MetaMode* non sécurisé vont respecter ces deux propriétés.

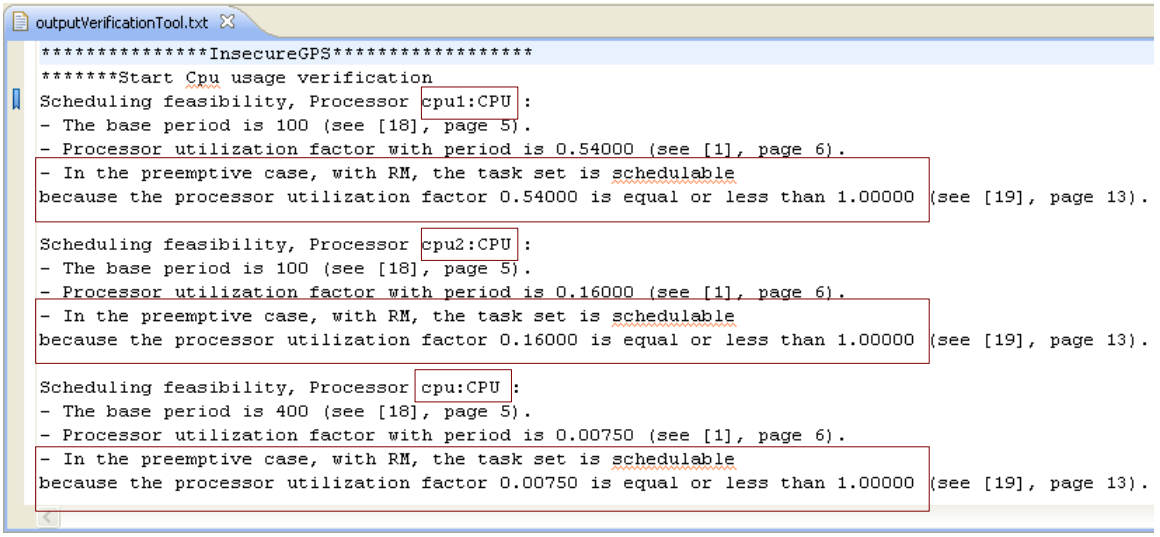

FIGURE 7.11 – Respect de la consommation CPU pour le *MetaMode* non sécurisé

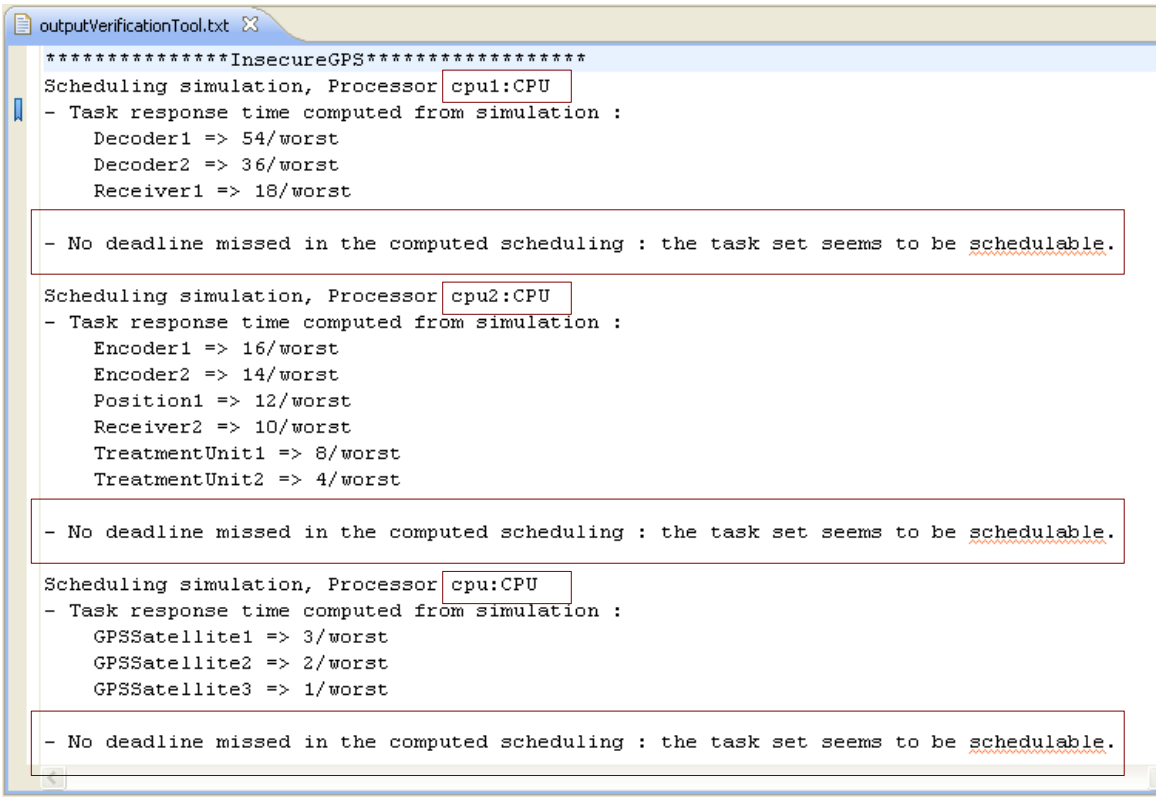

FIGURE 7.12 – Respect du respect des échéances pour le *MetaMode* non sécurisé

Structured Component	Nature	Period Deadline	WCET1 Processor 1Ghz	WCET2	memory size
Receiver	sporadic	100 ms	20 ms	2 ms	0.9 MB
Position	sporadic	100 ms	20 ms	2 ms	0.5 MB
TreatmentUnit	sporadic	100 ms	20 ms	4 ms	0.75 MB
Decoder	sporadic	100 ms	20 ms	2 ms	0.1 MB
Encoder	sporadic	100 ms	20 ms	2 ms	0.5 MB
GpsSatellite	periodic	400 ms	30 ms	0	0.9 MB
GpsControlBase	periodic	400 ms	30 ms	0	0.9 MB

TABLE 7.2 – Nouvelles Propriétés non-fonctionnelles des composants structurés du *MetaMode* non sécurisé

7.4.3 Modèle du système vérifié

Le tableau 7.2 représente les nouvelles propriétés non-fonctionnelles, ainsi que les caractéristiques de la nouvelle architecture du système pour le *MetaMode* non sécurisé. Pour les connexions logicielles, nous avons rectifié la valeur de la bande passante de départ à 5 b/s.

7.5 Contribution à la construction automatique du système MyGPS

La génération des modes fonctionne indépendamment de la génération de l'implantation du système. Dans la suite, nous allons montrer les deux types de génération. Tout d'abord, nous présentons la création des modes conformes aux politiques de reconfiguration spécifiées au niveau modèle pour configurer l'intergiciel. Ensuite, nous montrons la génération du code du système reconfigurable pour cet intergiciel.

7.5.1 Génération des modes

Le plugin de génération des modes permet d'analyser et de sélectionner les modes conformes aux politiques de reconfiguration, à partir des modèles spécifiant la reconfiguration (résultat de la phase de modélisation). Le résultat de cette analyse pour l'exemple MyGPS montre que sur tous les modes possibles, à partir du modèle, il n'y a que sept modes qui respectent les politiques de reconfiguration spécifiées : quatre modes pour le *MetaMode* non sécurisé et trois modes pour le MetaMode sécurisé sont sélectionnés. Ces modes sont donc ajoutés dans l'intergiciel (section 7.6). La Figure 7.13 montre que le *mode 4* du MetaMode non sécurisé respecte les politiques

de reconfiguration et donc il sera ajouté à l'intergiciel. Tandis que le *mode 5* ne respecte pas ces politiques et il sera donc éliminé.

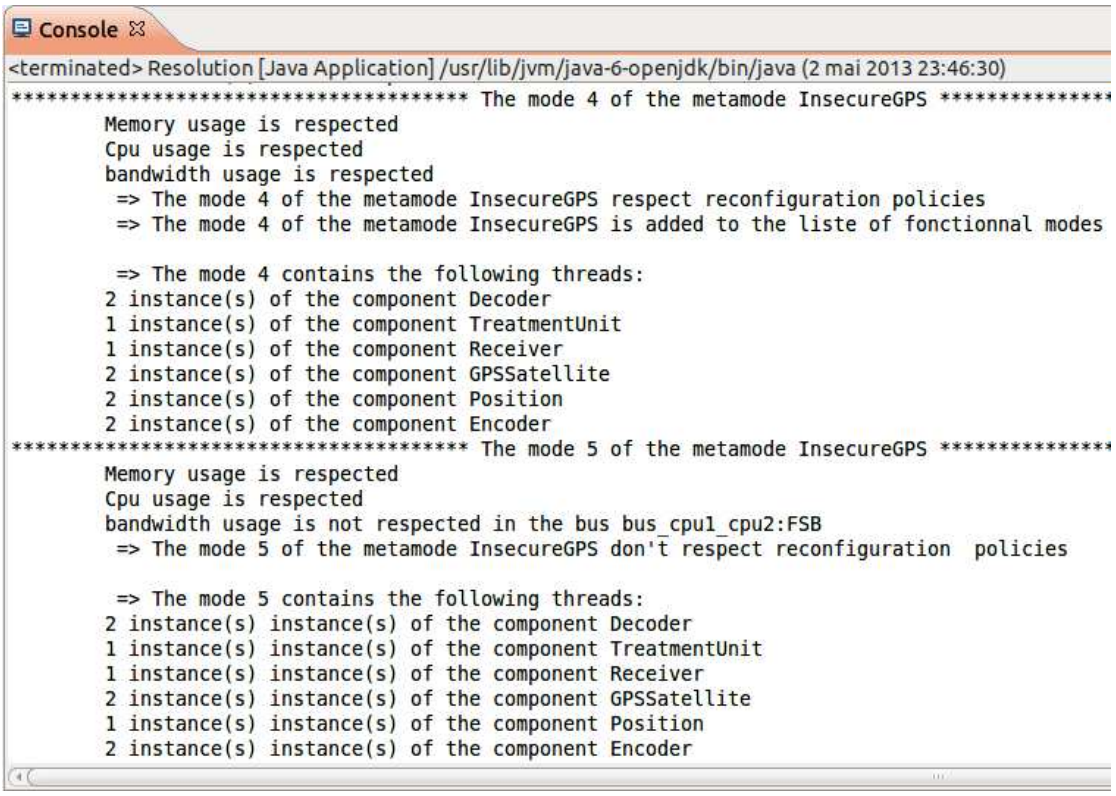

FIGURE 7.13 – Génération des modes respectant les politiques de reconfiguration

7.5.2 Génération du modèle d'implantation

Le plugin d'implantation est composé de deux types de générateurs : (1) des transformations M2M pour générer le modèle d'implantation et des transformations M2T pour générer le code Java.

La transformation M2M du modèle du système MyGPS permet de générer un fichier XMI (figure 7.14) contenant un paquetage système nommé GPS et trois sous paquetages. Ces sous paquetages représentent l'implantation des trois nœuds du système (GPSTerminal, GPSSattelite et GPSControlBase). Dans chaque paquetage, nous générons des classes pour implanter les composants qui seront alloués sur ces noeuds. Par exemple, pour le paquetage GPSTerminal nous générons les classes : *TransportHighLevelImpl*, *Deployment*, *Activity*, etc.

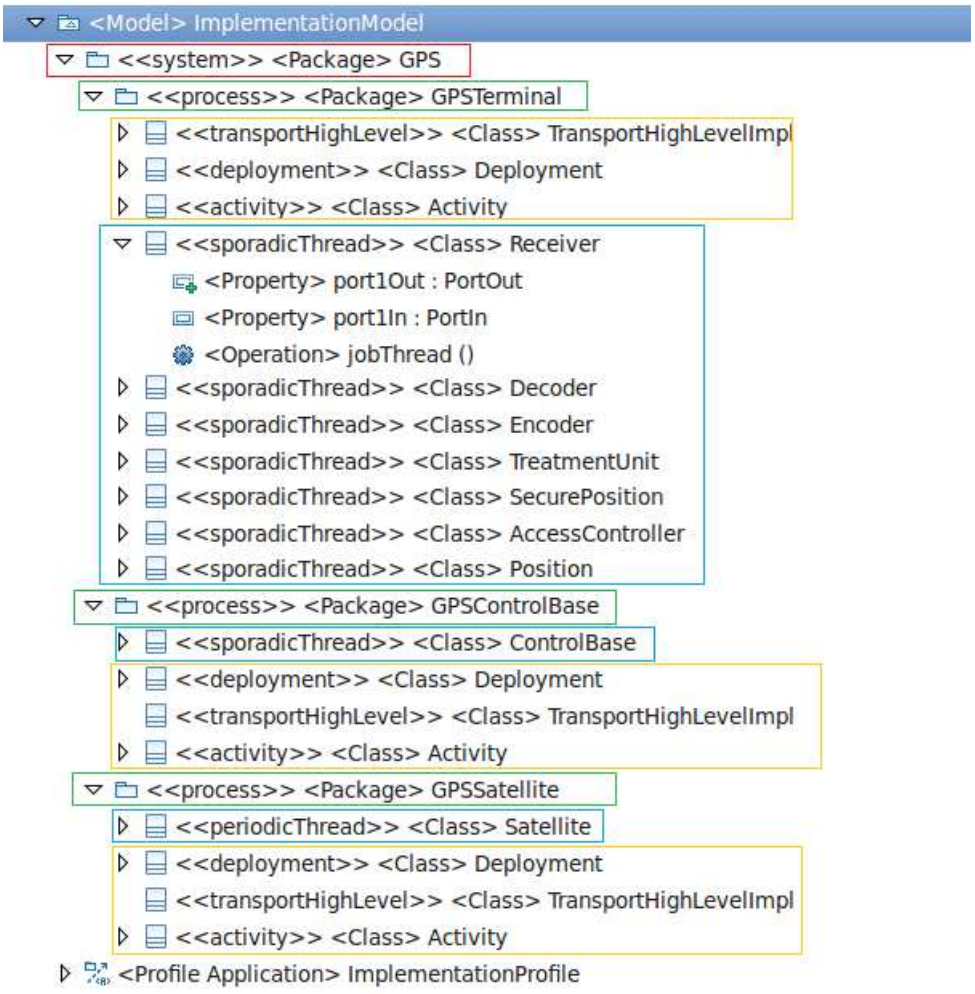

FIGURE 7.14 – Modèle d'implantation généré du système MyGPS

7.5.3 Génération de code

A partir du modèle d'implantation et en utilisant Acceleo, une partie du code du système MyGPS est générée. Le Listing 7.1 représente une partie du code de la classe Java *Position* générée. Cette classe importe des classes de l'intergiciel développé.

```
1
2  /**
3  * Generated with MTL : Acceleo
4  */
5  package GPSTerminal;
6
7  // Start of user code for imports
8
9  import fr.enst.ocarina.polyORB_HI_runtime.Context;
```

```
10  import fr.enst.ocarina.polyORB_HI_runtime.Debug;
11  import fr.enst.ocarina.polyORB_HI_runtime.Entry;
12  import fr.enst.ocarina.polyORB_HI_runtime.EventHandler;
13  import fr.enst.ocarina.polyORB_HI_runtime.InPort;
14  import fr.enst.ocarina.polyORB_HI_runtime.OutPort;
15  import fr.enst.ocarina.polyORB_HI_runtime.Port;
16  import fr.enst.ocarina.polyORB_HI_runtime.PortsRouter;
17  import fr.enst.ocarina.polyORB_HI_runtime.ProgramException;
18  import fr.enst.ocarina.polyORB_HI_runtime.Property;
19  import fr.enst.ocarina.polyORB_HI_runtime.SporadicTask;
20  import fr.enst.ocarina.polyORB_HI_runtime.Utils;
21  import fr.enst.ocarina.polyORB_HI_runtime.Utils.TimeUnit;
22  import java.util.Vector;
23
24  // End of user code
25      /**
26      * @author
27      */
28  public class Position {
29
30  \textbf{....}
31
32      // Position job
33      public void PositionJob(InPort inPort) throws ProgramException {
34      // Start of user code for operation jobThread
35      // TODO should be implemented
36      return null;
37      // End of user code
38      }
39  public void initialisation() {
40          // Creation of the Position's data output port.
41          Position_Out = new OutPort(idPortOut, Port.OUT_DATA_PORT,
42                  destPosition);
43          // Creation of the Position me event data input port's parameters.
44          GeneratedTypes defaultValuePosition = new GeneratedTypes(0);
45          Entry defaultEntryPosition = new Entry(defaultValuePosition);
46          // Creation of the PositionsourcePosition = new Vector();ary Position me's event data input
47          // port.
48          Position_In = new InPort(idPortIn, defaultEntryPosition,
49                  Port.IN_EVENT_DATA_PORT, 16, 0, sourcePosition);
50          positionTask = new SporadicTask(id, 100,
51                   TimeUnit.MilliSecond, 100, TimeUnit.MilliSecond, Utils.DEFAULT_PRIORITY,
52                   100000, new PositionHandler());
53          Entry[] PositionEntries = new Entry[16];
54          for (int i = 0; i < PositionEntries.length; i++) {
55              GeneratedTypes typePosition = new GeneratedTypes(0);
56              PositionEntries[i] = new Entry(typePosition);
57          }
58          PositionRouter = new PortsRouter(positionTask, PositionEntries, 1, 0);
59          Position_Out.setPortsRouter(PositionRouter);
60      }
61      public Position(int id, int idPortOut, int idPortIn, Vector destPosition , Vector sourcePosition){
62          this.destPosition = destPosition;
63          this.sourcePosition = sourcePosition;
```

```
64          this.idPortOut = idPortOut;
65          this.idPortIn = idPortIn;
66          initialisation();
67          P = new Property[7];
68          P[0]= new Property("id", id);
69          P[1]= new Property("priority", priority);
70          P[2]= new Property("WCET1", 0);
71          P[3]= new Property("WCET2", 0);
72          P[4]= new Property("stackSize", stackSize);
73          P[5]= new Property("period", period);
74          P[6]= new Property("deadLine", deadLine);
75      }
76  }
```

Listing 7.1 – Partie de la classe *Position* générée

7.6 Exécution de MyGPS sur l'intergiciel RCES4RTES

Dans ce qui suit, nous présentons une simulation de l'exécution de l'étude de cas MyGPS.

7.6.1 Configuration de l'intergiciel RCES4RTES

La phase de génération a permis de configurer l'intergiciel RCES4RTES. La classe *Deployment* générée permet d'initialiser le système par l'ajout des détails (noeuds, composants, connexions, etc) de la configuration initiale. Tandis que la génération des modes permet d'ajouter les modes respectant les politiques de reconfiguration. Le Listing 7.2 présente une partie de la classe *Deployment* qui permet de configurer les structures de données de la classe *Context* de l'intergiciel.

```
1  /**
2  * Generated with MTL : Acceleo
3  */
4  package GPSTerminal;
5
6  // Start of user code for imports
7  import fr.enst.ocarina.polyORB_HI_runtime.Context;
8  import fr.enst.ocarina.polyORB_HI_runtime.Event;
9  import fr.enst.ocarina.polyORB_HI_runtime.MetaMode;
10 import fr.enst.ocarina.polyORB_HI_runtime.Mode;
11 import fr.enst.ocarina.polyORB_HI_runtime.ThreadConnexion;
12 import fr.enst.ocarina.polyORB_HI_runtime.Suspenders;
13 import fr.enst.ocarina.polyORB_HI_runtime.TransportLowLevelSockets;
14
15 // End of user code
```

```
16      /**
17      * @author
18      */
19  public class Deployment {
20      \textbf{...}
21
22      //Initialize application's parameters.
23      public static void initialization() {
24          Context.nbNodes = 3;
25          Context.nbLocalEntities = 9;
26          Context.maxPayloadSize = 1024;// remplacer cette valeur en tenant compte
27          Context.myNode = terminalNode;
28          Context.myMode = 0;
29
30          Context.myNodesC = new Vector();
31          Context.myNodesC.add(satelliteNode);
32
33          Context.Events = new Vector();
34          Context.Events.add(new Event(0, 0, 1));
35          Context.Events.add(new Event(1, 1, 0));
36
37          Context.MetaModes = new Vector();
38          Context.MetaModes.add(new MetaMode(0, "metamodeInsecure"));
39          Context.MetaModes.add(new MetaMode(1, "metamodeInsecure"));
40
41          \textbf{...}
42          Context.types = new Vector();
43          Context.types.add("position");
44          Naming.initialization();
45          Context.transport = new TransportHighLevelImpl();
46          Context.entitiesOffset = 0;
47          Suspenders.initialization();
48          TransportLowLevelSockets.initialize();
49          initialOperationalMode(Context.myMode);
50
51      }
52
53      public static void initialOperationalMode( int idMode ){
54
55          \textbf{...}
56          Context.threads = new Vector();
57          Position positionThread = new Position(position, pos_Source, pos_Sink,
58                  ((ThreadConnexion)mode.listThreadsConnexion.elementAt(indexSat)).listDest,
59                  ((ThreadConnexion)mode.listThreadsConnexion.elementAt(indexSat)).listSource);
60          Context.threads.add(positionThread);
61
62          Context.destinationsTable = new Hashtable();
63          Context.destinationsTable.put(positionThread.idPortOut, positionThread.Position_Out.
              destinations);
64
65
66          Context.typesTable = new Hashtable();
67          Context.typesTable.put(position,"position");
68          Context.typesTable.put(satellite,"satellite");
```

```
69          Context.typesTable.put(position, "position");
70          Context.typesTable.put(receiver1, "receiver");
71          Context.typesTable.put(decoder, "decoder");
72          Context.typesTable.put(treatment, "treatmentUnit");
73          Context.typesTable.put(encoder, "encoder");
74          Context.typesTable.put(contBase, "controlBase");
75
76          Context.entitiesTable = new Hashtable();
77          Context.entitiesTable.put(satellite, satelliteNode);
78          Context.entitiesTable.put(position, terminalNode);
79          Context.entitiesTable.put(receiver1, terminalNode);
80          Context.entitiesTable.put(decoder, terminalNode);
81          Context.entitiesTable.put(treatment, terminalNode);
82          Context.entitiesTable.put(encoder, terminalNode);
83          Context.entitiesTable.put(contBase, controlBaseNode);
84
85          Context.portsTable = new Hashtable();
86          Context.portsTable.put(pos_Sink,position);
87          Context.portsTable.put(pos_Source,position);
88          Context.portsTable.put(rec1_Sink, receiver1);
89          Context.portsTable.put(rec1_Source, receiver1);
90          Context.portsTable.put(rec2_Sink, receiver2);
91          Context.portsTable.put(rec2_Source, receiver2);
92          Context.portsTable.put(enc_Sink, encoder);
93          Context.portsTable.put(enc_Source, encoder);
94          Context.portsTable.put(dec_Sink, decoder);
95          Context.portsTable.put(dec_Source, decoder);
96          Context.portsTable.put(treat_Sink, treatment);
97          Context.portsTable.put(treat_Source, treatment);
98          Context.portsTable.put(contBase_Sink, contBase);
99          Context.portsTable.put(contBase_Source, contBase);
100         Context.portsTable.put(sat_Sink, satellite);
101         Context.portsTable.put(sat_Source1, satellite);
102         Context.portsTable.put(sat_Source2, satellite);
103
104         Context.reconfEntity = new Hashtable();
105         Context.reconfEntity.put(reconfSatellite, satelliteNode);
106         Context.reconfEntity.put(reconfContBase, controlBaseNode);
107         Context.reconfEntity.put(reconfTerminal, terminalNode);
108
109         Context.listInPort = new Vector();
110         Context.listInPort.add(positionThread.Position_In);
111         Context.listInPort.add(rec1_Sink);
112         Context.listInPort.add(rec2_Sink);
113         Context.listInPort.add(pos_Sink);
114         Context.listInPort.add(accessCont_Sink);
115         Context.listInPort.add(treat_Sink);
116         Context.listInPort.add(enc_Sink);
117         Context.listInPort.add(dec_Sink);
118         Context.listInPort.add(contBase_Sink);
119         Context.listInPort.add(secuPos_Sink);
120
121         for(int i=0;i<Context.threads.size();i++){
122             if (Context.threads.elementAt(i) instanceof Position)
```

```
123                 ((Position) Context.threads.elementAt(i)).positionTask.start();
124             }
125             System.out.println("initial_configuration_is_completed");
126         }
127     }
```

Listing 7.2 – Partie de la classe *Deployment* générée

Après la génération du code et afin d'illustrer l'utilisation de l'intergiciel RCES4RTES développé, nous définissons un mode (configuration) pour chaque *MetaMode* du système MyGPS respectant les politiques de reconfiguration (généré pendant la phase de génération). La Figure 7.15 présente une configuration du MetaMode non sécurisé tandis que la Figure 7.16 représente une configuration du MetaMode sécurisé.

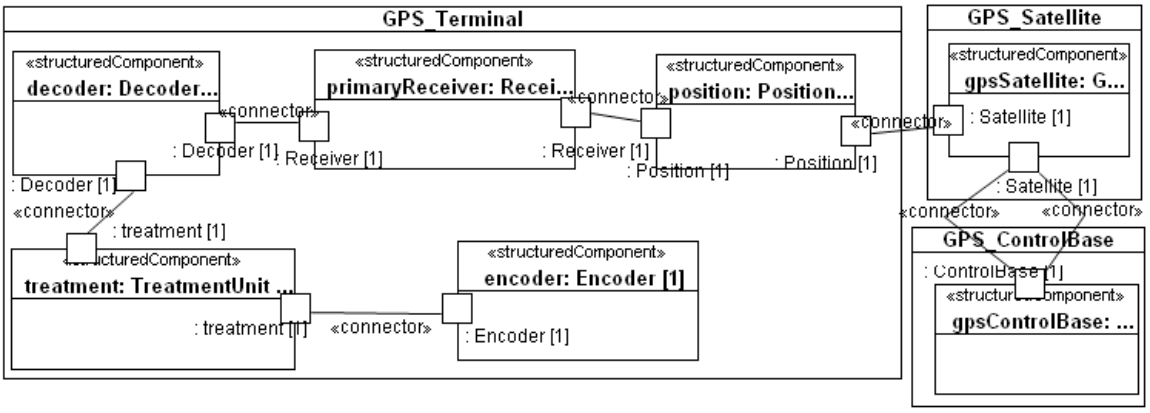

FIGURE 7.15 – Configuration non sécurisée du MyGPS

7.6.2 Trace d'exécution dans un mode non sécurisé

Nous montrons dans la figure 7.17 une trace d'exécution du terminal GPS pour un mode non sécurisé. Tout d'abord, pour le noeud Terminal, nous commençons par la création des instances des composants *Position*, *Receiver*, etc. Après leur création, les instances sont en état d'attente d'événement. Par exemple, le noeud Terminal a reçu des informations du noeud Satellite.

7.6.3 Exemple de reconfiguration

Nous illustrons la reconfiguration par un passage d'un mode non sécurisé (figure 7.15) vers un mode sécurisé (Figure 7.16) du système MyGPS. Le passage

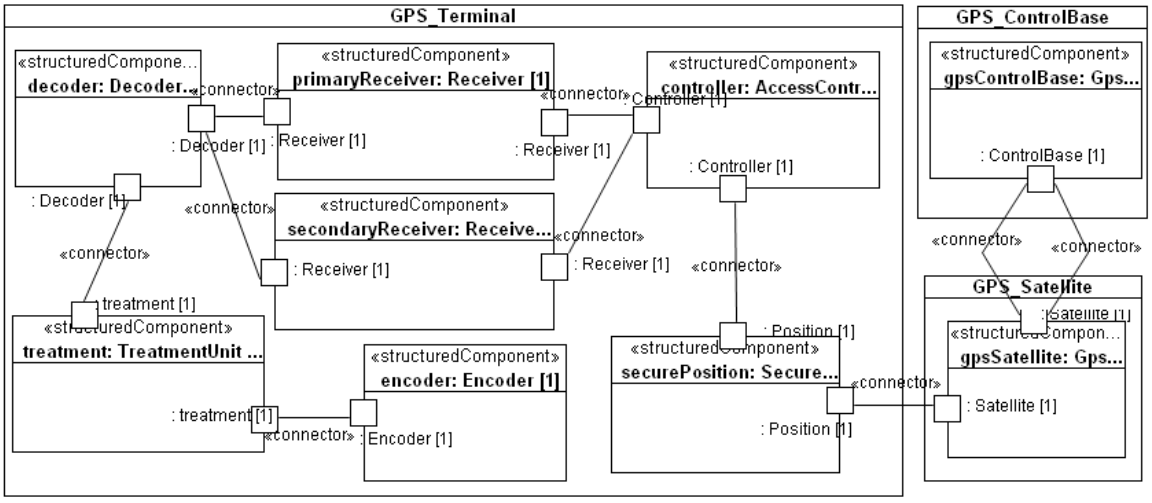

FIGURE 7.16 – Configuration sécurisée du MyGPS

```
Creation of the position instance on GPS_Terminal node
position : wait initialization
position : new Dispatch
position is waiting for incoming events
Creation of the primaryReceiver instance on GPS_Terminal node
primaryReceiver : wait initialization
primaryReceiver : new Dispatch
primaryReceiver is waiting for incoming events
Creation of the decoder instance on GPS_Terminal node
decoder : wait initialization
decoder : new Dispatch
decoder is waiting for incoming events
Creation of the treatmentUnit instance on GPS_Terminal node
treatmentUnit : wait initialization
Creation of the encoder instance on GPS_Terminal node
encoder : wait initialization
encoder : new Dispatch
treatmentUnit : new Dispatch
encoder is waiting for incoming events
treatmentUnit is waiting for incoming events
GPS_Terminal node : receives 18 bytes from GPS_Satellite node
position : store received message
```

FIGURE 7.17 – Trace de l'exécution du terminal du GPS dans le mode non sécurisé

d'une configuration d'un MetaMode non sécurisé vers une configuration d'un MetaMode sécurisé consiste à supprimer toutes les instances du composant structuré Position et à ajouter des instances des deux composants structurés *SecurePosition* et *AccessController*. Nous détaillons les actions à exécuter pour mettre en œuvre cette reconfiguration (figure 7.18) :

– ajouter l'instance *controller* du composant *AccessController*,

- ajouter l'instance *securePosition* du composant *SecurePosition*,
- ajouter l'instance *secondaryReceiver* du composant *Receiver*,
- ajouter une connexion entre *gpsSatellite* et *securePosition*,
- ajouter une connexion entre l'instance *securePosition* et l'instance *controller*,
- ajouter une connexion entre l'instance *controller* et l'instance *primaryReceiver*,
- ajouter une connexion entre l'instance *controller* et l'instance *secondaryReceiver*,
- ajouter une connexion entre l'instance *secondaryReceiver* et l'instance *decoder*,
- supprimer la connexion entre l'instance *position* et l'instance *primaryReceiver*,
- supprimer la connexion entre l'instance *position* et l'instance *gpsSatellite*,
- supprimer l'instance *position* du composant *Position*.

```
Dynamic reconfiguration: the securePosition whose type is SecurePosition has been added
on GPS_Terminal node
Dynamic reconfiguration: the controller whose type is AccessController has been added
on GPS_Terminal node
Dynamic reconfiguration: the secondaryReceiver whose type is Receiver has been added
on GPS_Terminal node
GPS_Terminal node: send 18 bytes to GPS_Satellite node
Dynamic reconfiguration: distant connection between the port 17 and 11 has been added
Dynamic reconfiguration: local connection between the port 10 and 13 has been added
Dynamic reconfiguration: local connection between the port 12 and 3 has been added
Dynamic reconfiguration: local connection between the port 12 and 15 has been added
Dynamic reconfiguration: local connection between the port 14 and 5 has been added
Creation of the securePosition instance on GPS_Terminal node
securePosition : wait initialization
securePosition : new Dispatch
securePosition is waiting for incoming events
Creation of the controller instance on GPS_Terminal node
Creation of the secondaryReceiver instance GPS_Terminal node
Coherence: the position has been locked
controller  : wait initialization
Consistency: the position instance has been unlocked
secondaryReceiver : wait initialization
Dynamic reconfiguration: local connection between the port 0 and 3 has been removed
controller  : new Dispatch
secondaryReceiver : new Dispatch
isconnection : existe=true
controller  is waiting for incoming events
secondaryReceiver is waiting for incoming events
GPS_Terminal node : send 18 bytes to GPS_Satellite node
Dynamic reconfiguration: distant connection between the port 17 and 1 has been removed
Dynamic reconfiguration: position has been removed from the GPS_Terminal node
```

FIGURE 7.18 – Fichier log de la reconfiguration du mode non sécurisé au mode sécurisé

7.6.4 Propriétés du système reconfigurable MyGPS

Nous allons maintenant montrer que l'intergiciel assure la cohérence du système MyGPS tout en respectant une faible empreinte mémoire et en offrant des mécanismes de supervision.

Après la transition du mode non sécurisé vers le mode sécurisé, le système est toujours cohérent, tout en préservant les contraintes temporelles. Nous observons dans la figure 7.19 deux instances *primaryReceiver* et *secondaryReceiver* du composant *Rceiver* s'exécutant normalement et respectant leurs échéances (100 ms). Par exemple, l'instance *secondaryReceiver* termine son exécution après 57 ms.

En outre, nous avons calculé l'empreinte mémoire du système MyGPS sur chaque nœud : 51.5 KB pour le nœud *Terminal*, 25.6 KB pour le nœud *Satellite* et 25.9 KB pour le nœud *ControlBase*. Etant donné la faible empreinte mémoire de l'intergiciel RCES4RTES ($\simeq$ 131 KB), nous pouvons conclure que l'empreinte mémoire de chaque nœud reste assez faible.

```
secondaryReceiver calls a sub program at (1321967893590 ms, 445144 ns)
primaryReceiver, getValue of input port 3
primaryReceiver, readIn : reading the oldest element in the queue of event [data] input por
primaryReceiver, readIn : value read from input port 3
primaryReceiver, readIn : reading the oldest element in the queue of event [data] input por
primaryReceiver, readIn : value read from input port 3
primaryReceiver, getValue done
primaryReceiver, nextValue for event [data] input port 3
primaryReceiver, dequeue : dequeuing event [data] input port 3
primaryReceiver, historyIncrementFirst : globalHistoryFirst = 4
primaryReceiver calls a sub program at (1321967893642 ms, 430784 ns)
secondaryReceiver  termine impl at (1321967893647 ms, 293352 ns)
 at t=14h18m13s647ms ***** the node 0 converted the signal into the value : 15
secondaryReceiver, storeOut : storing value for output port 14
secondaryReceiver, storeOut : value stored for output port 14
secondaryReceiver, sendOutput for output port 14
secondaryReceiver, setInvalid : setting invalid for output port 14
secondaryReceiver, sendOutput output port 14
secondaryReceiver, send output to input port 5 of entity 2
decoder : store received message
.....
decoder calls a sub program at (1321967893648 ms, 987482 ns)
primaryReceiver termine impl at (1321967893716 ms, 950898 ns)
```

FIGURE 7.19 – Respect des deadlines des tâches

7.7 Conclusion

Dans ce chapitre, nous avons cnsidéré un cas d'étude sur un système de localisation afin de démontrer la faisabilité de notre approche pour la modélisation, la vérification, la génération et l'exécution des systèmes TR^{2}E dynamiquement reconfigurables. Le système MyGPS est une version simplifié du GPS. Il est formé de 9 types de composants dont 2 sont variables, c.à.d qu'ils apparaissent ou non dans un mode du système. Nous avons spécifié deux MetaModes, une architecture logicielle, une architecture matérielle et une allocation. Ensuite, nous avons validé le modèle du système vis-à-vis des contraintes non-fonctionnelles de ses constituants. Ainsi, nous

avons généré les modes, conformes aux politiques de reconfiguration de MyGPS, spécifiées dans le modèle, pour configurer l'intergiciel RCES4RTES. Enfin, nous avons généré une implantation dédiée à cet intergiciel. Une simulation de l'exécution de MyGPS sur l'intergiciel RCES4RTES montre la réalité de son implémentation tout en ouvrant d'autres pistes pour de futures réalisations et utilisations.

Chapitre 8

Conclusion et Perspectives

Pour conclure ce document, nous présentons un bilan de nos principales contributions, comme le montre la Figure 8.1 qui reprend l'ensemble des phases de la méthodologie et les outils proposés pour sa mise en œuvre. Nous développons ensuite les perspectives.

8.1 Rappel de la problématique

De part l'évolution exponentielle des exigences des utilisateurs mais aussi la variation des contraintes de l'environnement d'exécution, les systèmes embarqués temps réel répartis (TR^2E) deviennent de plus en plus importants et demandés dans notre vie quotidienne. Par conséquent, les développeurs de ces systèmes doivent optimiser leurs temps de mise sur le marché (*time to market*) tout en fournissant des produits novateurs, compétitifs et à faible coût. Ainsi, il est nécessaire de concevoir des outils logiciels, des méthodologies pour accompagner les développeurs de ces systèmes.

La reconfiguration dynamique complique considérablement le développement et l'exécution de ces systèmes. Ces derniers ont en effet des contraintes temporelles et de ressources strictes et des exigences de préservation des propriétés non-fonctionnelles (telle que la consommation CPU) durant les reconfigurations. La vérification de ces propriétés au cours de l'exécution est ardue. Ceci parce qu'elle nécessite des ressources de calcul supplémentaires, déjà très limitées dans les systèmes embarqués. De plus, elle nécessite un temps d'exécution additionnel qui peut poser un problème pour le fonctionnement des systèmes critiques.

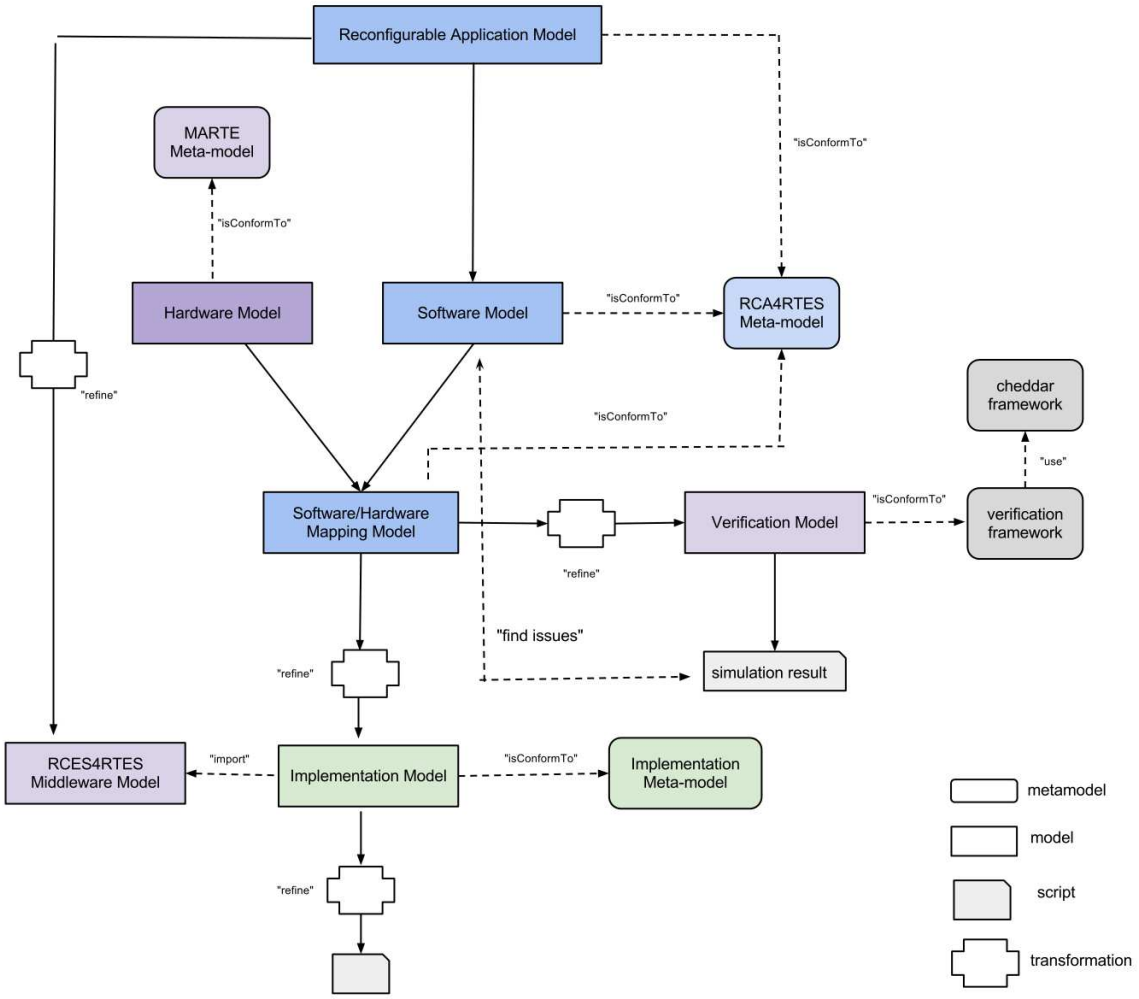

FIGURE 8.1 – Framework de développement de systèmes TR^{2}E reconfigurables

8.2 Approche proposée

Notre travail est centré sur la spécification de la reconfiguration pour les systèmes temps réel, sur la proposition d'une méthodologie, ainsi que sa mise en œuvre dans le cadre de l'ingénierie dirigée par les modèles. Plus précisément, nous avons proposé dans cette thèse des langages de modélisation, des transformations, une plate-forme d'exécution (intergiciel), ainsi qu'un processus pour gouverner le développement de ces systèmes.

L'approche que nous proposons est basée sur la notion de MetaMode (extension de machine à état UML). Un MetaMode représente une classe de comportements (configurations) possibles du système, associée à une architecture logicielle et matérielle.

Préalablement à l'élaboration de ce processus, nous avons construit une plateforme d'exécution dédiée aux systèmes TR^{2}E dynamiquement reconfigurables. En effet, nous avons élaboré un intergiciel nommé RCES4RTES basé sur PolyORB_HI offrant un ensemble de routines facilitant le développement de ces systèmes.

Le processus de construction des systèmes TR^{2}E dynamiquement reconfigurables est composé de trois phases principales : une phase de modélisation, une phase de vérification et une phase de génération. Durant la phase de modélisation, le système reconfigurable est décrit en spécifiant les reconfigurations dynamiques à travers une machine à états composée des *MetaModes* et de transitions entre eux. Une transition représente un événement qui déclenche une reconfiguration du *MetaMode* source vers un *MetaMode* cible. En outre, le système est soumis à un ensemble de politiques de reconfigurations (consommation CPU, mémoire et bande passante). La description de la machine à états et des politiques de reconfiguration est réalisée en utilisant le profil RCA4RTES. Chaque *MetaMode* est modélisé à travers une approche à base de composants : RCA4RTES pour la partie logicielle et l'allocation et MARTE pour la partie matérielle.

La phase de vérification consiste à s'assurer que les propriétés non-fonctionnelles (respect des échéances des tâches, consommation mémoire et consommation de la bande passante) spécifiées au niveau modèle de conception (phase précédente) sont bien satisfaites. Le cas échéant, la conception sera alors rectifiée. La vérification est réalisée par un plugin basé sur le framework cheddar. Lorsque tous les tests sont réussis, nous pouvons passer à la phase suivante dite de génération.

La phase de génération est composée de deux sous-phases : la génération des modes et la génération du code. Les générations sont spécifiées via des transformations de modèles. Ainsi, nous définissons deux types de transformations :

- M2M qui consiste à transformer les modèles spécifiés durant la phase de modélisation vers des modèles d'implantation.
- M2T qui consiste à transformer les modèles d'implantation vers un code exécutable, dans notre cas "Java".

La génération des modes consiste à identifier l'ensemble des configurations (ou les modes) des *MetaModes* qui respectent les politiques de reconfiguration. Les modes ainsi créés sont utilisés pour configurer l'intergiciel (RCES4RTES) par l'ajout de ces derniers dans la classe *Context* de l'intergiciel. Enfin, le code du système à exécuter sur l'intergiciel est généré.

8.3 Discussion

Dans le cadre de cette thèse, nous avons proposé un processus de développement outillé permettant de concevoir des systèmes TR^{2}E dynamiquement reconfigurables. Ce processus supporte les systèmes TR^{2}E dynamiquement reconfigurables contrairement à OCARINA [37] et ModES [9] qui ne supportent pas des systèmes dynamiquement reconfigurables. Les deux processus de développement COMDES [19] et TimeAdapt [15] permettant de concevoir des systèmes répartis reconfigurables ne proposent pas d'intergiciel facilitant la génération du code comme c'est le cas de notre processus.

Notre processus de développement offre de nouveaux concepts permettant la modélisation des systèmes TR^{2}E et en particulier les reconfigurations dynamiques. Contrairement à MARTE [45] et AADL [52] qui permettent la modélisation des systèmes embarqués temps réel dynamiquement reconfigurables en énumérant toutes les configurations possibles, notre approche offre de nouveaux concepts permettant de caractériser les configurations au lieu de spécifier chacune d'entre elles. Pour remédier aux problèmes soulevés par la nécessité d'énumérer toutes les configurations possibles d'un système, le concept de *MetaMode* a été introduit. Les reconfigurations dynamiques sont donc spécifiées par des machines à états composées par des *MetaModes* et des transitions entre eux. L'utilisation des *MetaModes* a plusieurs avantages. Elle permet de gagner du temps en évitant l'énumération de tous les modes possibles et de couvrir tous les modes possible du systèmes facilement. En utilisant les *MetaModes*, il sera simple de Modéliser les systèmes ayant un grand nombre de modes fonctionnels. L'utilisateur peut facilement modéliser ces modes à travers les *MetaModes* en minimisant le risque d'oublier des modes fonctionnels. A partir des *MetaModes*, les modes fonctionnels respectant les politiques de reconfiguration seront générés automatiquement. L'utilisateur n'est pas censé chercher et calculer à chaque fois les modes fonctionnels exigés et respectant les politiques. Cependant, l'identification des *MetaModes* du système nécessite une réflexion.

La plupart des approches de vérification existantes assurent la vérification au cours de l'exécution. Par contre, notre processus assure la vérification de certaines propriétés non-fonctionnelles (consommation CPU, respect des échéances des tâches, consommation mémoire, consommation de la bande passante et absence d'interblocage et de famine) au niveau modèle de conception. Contrairement aux frameworks tels que VERTAF [22] et Tasm Toolset [49] assurant la vérification au niveau modèle pour des systèmes statiques, notre approche assure la vérification pour des systèmes

répartis dynamiquement reconfigurables. De plus et contrairement à Cheddar [56] assurant la vérification de respect des échéances et de la consommation CPU pour des tâches périodiques seulement, elle supporte des systèmes ayant des tâches périodiques, sporadiques et hybrides.

Afin de faciliter la génération de code, un intergiciel dédié aux systèmes TR^{2}E dynamiquement reconfigurables a été développé. Il a une faible empreinte mémoire comparé aux intergiciels existants dédiés pour des systèmes embarqués dynamiquement reconfigurables tels que CIAO [58] et FLARe [4]. Contrairement aux intergiciels DynamicTAO [26] et SwapCIAO [3], notre intergiciel assure la reconfiguration dynamique ainsi que la supervision et la cohérence pour ces systèmes. La phase de génération consiste à configurer l'intergiciel et à générer le code.

Cependant, notre processus ne prend pas en considération les effets d'une transition d'un mode vers un autre dans ses différentes phases :

- Ainsi, durant la phase de vérification, il ne teste pas la consommation des ressources et les contraintes temporelles reliées aux tâches de reconfiguration.
- Si l'une des propriétés n'est pas satisfaite, le concepteur doit rectifier la phase de modélisation. Ainsi, notre processus ne propose pas d'autres alternatives (modèles rectifiés) en cas d'échec.
- Ce processus ne prend pas en compte la qualité de service, la tolérance aux fautes, l'efficacité et la consommation d'énergie.

Par ailleurs, nous proposons une approche de reconfiguration dirigée par les modèles dont l'architecture générée n'est pas suffisamment flexible pour supporter les reconfigurations automatiques sur des systèmes embarqués hétérogènes. En outre, la validation de la méthodologie et de son outillage a été réalisée sur un cas d'étude simple (GPS). Il serait judicieux dans le futur d'envisager des cas d'utilisation industriels.

8.4 Perspectives

Le travail que nous avons réalisé pour la modélisation, la vérification, les plateformes d'exécution et la suite d'outils représente en fait une première étape qui ouvre la voie à plusieurs perspectives de recherche.

Ainsi, nous visons à rédiger un guide méthodologique pour le développement des systèmes TR^{2}E dynamiquement reconfigurables fondé sur l'ensemble des outils que nous venons de proposer. En cas d'échec de certains tests, il sera proposé des alter-

natives pour assister le concepteur. Ceci permettra alors de simplifier et d'améliorer le processus.

Dans cette thèse, nous avons assuré la vérification niveau modèle afin de générer le code correctement. En effet, nous nous intéressons à concevoir des systèmes TR^2E dynamiquement reconfigurables avec une simple étape de vérification. Aussi, nous étudions la possibilité d'améliorer la phase de vérification. Ainsi, nous visons à optimiser les algorithmes de vérification et à considérer les effets de la reconfiguration dynamique (les transitions entre les modes). De plus, nous cherchons à explorer d'autres propriétés non-fonctionnelles à vérifier.

Pour augmenter la réutilisation des bonnes pratiques liées à la reconfiguration, nous préconisons l'étude et la formalisation d'un ensemble de patrons de conception pour la reconfiguration. Ainsi, ces patrons seront ensuite interprétés dans le contexte du méta-modèle RCA4RTES pour pouvoir réutiliser le processus, les outils et la plateforme d'exécution proposés.

Pour l'analyse et la vérification, nous souhaitons utiliser les méthodes formelles pour compléter les langages de modélisation proposés. En effet, capturer les nouveaux concepts de modélisation introduits dans ce travail de thèse par un langage formel, comme par exemple le langage Z, permettra d'une part d'ajouter de la sémantique aux modèles utilisés dans le processus, mais aussi d'aider à assurer leur correction par construction.

Publications de Fatma KRICHEN

1. Fatma Krichen, Brahim Hamid, Bechir Zalila, Mohamed Jmaiel, and Bernard Coulette. Development of Reconfigurable Distributed Embedded Systems with a Model-Driven Approach. *Journal of Concurrency and Computation : Practice and Experience*, 2013. To appear.

2. Fatma Krichen, Amal Ghorbel, Brahim Hamid, and Bechir Zalila. An MDE-Based Approach for Reconfigurable Embedded Systems. In *Proceedings of the 21st IEEE International Conference on Collaboration Technologies and Infrastructures*, pages 78–83, Toulouse, France, juin 2012. IEEE Computer Society.

3. Fatma Krichen, Bechir Zalila, Mohamed Jmaiel, and Brahim Hamid. A Middleware for Reconfigurable Distributed Real-Time Embedded Systems. In *Proceedings of the ACIS International Conference on Software Engineering Research, Management and Applications SERA (selected papers)*, Studies in Computational Intelligence, pages 81–96, Shanghai, China, 2012. Springer.

4. Fatma Krichen, Brahim Hamid, Bechir Zalila, and Mohamed Jmaiel. Design-Time Verification of Reconfigurable Real-Time Embedded Systems. In *Proceedings of the 9th IEEE International Conference on Embedded Software and Systems*, pages 1487–1494, Liverpool, UK, june 2012. IEEE.

5. Fatma Krichen, Amal Gassara, Bechir Zalila, and Mohamed Jmaiel. Towards a Verification Approach for Reconfigurable Embedded Systems. In *Proceedings of the 17th IEEE Symposium on Computers and Communications*, pages 750–752, Cappadocia, Turkey, july 2012. IEEE.

6. Fatma Krichen, Amal Gassara, Bechir Zalila, Brahim Hamid, and Mohamed Jmaiel. Modélisation et Vérification des Systèmes Embarqués Temps Réel Reconfigurables. In *6ème Conférence Internationale Francophone sur les Architectures Logicielles (CAL)*, 2012.

7. Fatma Krichen, Brahim Hamid, Bechir Zalila, and Mohamed Jmaiel. Towards a Model-Based Approach for Reconfigurable DRE Systems. In *Proceedings of the 5th European Conference on Software Architecture*, pages 295–302, Essen, Germany, September 2011. Springer.

8. Brahim Hamid and Fatma Krichen. Model-based engineering for dynamic reconfiguration in DRTES. In *Proceedings of the Fourth European Conference on Software Architecture. WORKSHOP SESSION : VIII Nordic Workshop*

on Model-Driven Software Engineering, pages 269–276, New York, NY, USA, 2010. ACM.

9. Fatma Krichen. Position paper : Advances in Reconfigurable Distributed Real Time Embedded Systems. In *International workshop on Distributed Architecture modeling for Novel component based Embedded systems*, pages 273–278, Tozeur, Tunisia, May 2010. IEEE.

10. Fatma Krichen, Brahim Hamid, Bechir Zalila, and Bernard Coulette. Designing Dynamic Reconfiguration of Distributed Real Time Embedded Systems. In *Proceedings of the 10th annual international conference on New Technologies of Distributed Systems*, pages 249–254, Tozeur, Tunisie, June 2010. IEEE Computer Society.

Bibliographie

[1] String template. http ://www.stringtemplate.org/. 35

[2] Tobias Amnell, Gerd Behrmann, Johan Bengtsson, Pedro R. D'Argenio, Alexandre David, Ansgar Fehnker, Thomas Hune, Bertrand Jeannet, Kim Guldstrand Larsen, M. Oliver Möller, Paul Pettersson, Carsten Weise, and Wang Yi. Uppaal - now, next, and future. In *Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes*, pages 99–124, London, UK, UK, 2001. Springer-Verlag. 20, 21

[3] Jaiganesh Balasubramanian, Balachandran Natarajan, Douglas C. Schmidt, Aniruddha S. Gokhale, Jeff Parsons, and Gan Deng. Middleware Support for Dynamic Component Updating. In *Proceedings of the OTM Conferences (2)*, pages 978–996. Springer, 2005. 4, 27, 139

[4] Jaiganesh Balasubramanian, Sumant Tambe, Chenyang Lu, Aniruddha S. Gokhale, Christopher D. Gill, and Douglas C. Schmidt. Adaptive Failover for Real-Time Middleware with Passive Replication. In *Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 118–127. IEEE, 2009. 4, 28, 139

[5] Eric J. Bruno and Greg Bollella. *The Real-Time Specification for Java*, chapter Real-Time Java^{tm} Programming with Java RTS. Prentice Hall, 2009. 84

[6] Alan Burns. The Ravenscar Profile. *Ada Lett.*, XIX :49–52, December 1999. 28, 33, 64, 73

[7] Alan Burns, Brian Dobbing, and Tullio Vardanega. Guide for the use of the Ada Ravenscar Profile in high integrity systems. *Ada Lett.*, XXIV :1–74, June 2004. 46

[8] W. El Hajj Chehade, Ansgar Radermacher, François Terrier, Bran Selic, and Sébastien Gérard. A model-driven framework for the development of portable

real-time embedded systems. In *Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems*, pages 45–54, Las Vegas, Nevada, USA, 2011. IEEE Computer Society. viii, 31, 37

[9] Francisco Assis M. do Nascimento, Marcio F. S. Oliveira, and Flavio Rech Wagner. Modes : Embedded systems design methodology and tools based on mde. In *Proceedings of the Fourth International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, pages 67–76, Washington, DC, USA, 2007. IEEE Computer Society. 30, 37, 138

[10] Francisco Assis Moreira do Nascimento, Marcio Ferreira da Silva Oliveira, and Flavio Rech Wagner. Formal Verification for Embedded Systems Design Based on MDE. In *Proceedings of the 10th International Embedded Systems Symposium*, pages 159–170, Langenargen, Germany, September 2009. Springer. 4, 20, 24, 25

[11] C. Schmidt Douglas and Krishnakumar Balasubramanian. *Model Driven Engineering for Distributed Real-time Embedded Systems*, chapter Model-Driven Development of Distributed Real-time and Embedded Systems. Wiley, 2005. 32

[12] Schmidt Douglas, Gokhale Aniruddha, Natarajan Balachandran, Neema Sandeep, Bapty Ted, Parsons Jeff, Gray Jeff, Nechypurenko Andrey, and Wang Nanbor. Cosmic : An mda generative tool for distributed real-time and embedded component middleware and applications. In *Proceedings Of The Oopsla 2002 Workshop On Generative Techniques In The Context Of Model Driven Architecture*. ACM, 2003. 32, 37

[13] ECSS-E-ST-50-12C. SpaceWire - Links, Nodes, Routers and Networks. European Space Agency, July 2008. Technical report. 64, 77

[14] Madeleine Faugère, Thimothée Bourbeau, Robert de Simone, and Sébastien Gérard. MARTE : Also an UML Profile for Modeling AADL Applications. In *Proceedings of the International Conference on Engineering of Complex Computer Systems*, pages 359–364. IEEE, 2007. 15

[15] S. Fritsch and S. Clarke. Timeadapt : timely execution of dynamic software reconfigurations. In *Proceedings of the 5th Middleware doctoral symposium*, pages 13–18, New York, NY, USA, 2008. ACM. 34, 37, 138

[16] Roman Gumzej, Matjaz Colnaric, and Wolfgang A. Halang. Safe and Timely Scenario Switching in UML Real-Time Projects. In *Proc. of the International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 335–342. IEEE, 2006. 12

[17] Roman Gumzej, Matjaz Colnaric, and Wolfgang A. Halang. A reconfiguration pattern for distributed embedded systems. *Software and System Modeling*, 8(1) :145–161, 2009. 12, 15

[18] Roman Gumzej and Wolfgang A. Halang. A safety shell for uml-rt projects. In *Proceedings of the International Multiconference on Computer Science and Information Technology*, pages 629–632, Wisla, Poland, 2008. IEEE. 13

[19] Y. Guo, K. Sierszecki, and C. Angelov. A (re)configuration mechanism for resource-constrained embedded systems. In *Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference*, pages 1315–1320, Washington, DC, USA, 2008. IEEE Computer Society. 36, 37, 138

[20] Brahim Hamid and Fatma Krichen. Model-based engineering for dynamic reconfiguration in DRTES. In *Proceedings of the Fourth European Conference on Software Architecture. WORKSHOP SESSION : VIII Nordic Workshop on Model-Driven Software Engineering*, pages 269–276, New York, NY, USA, 2010. ACM. 45

[21] Tom Henzinger and Joseph Sifakis. The embedded systems design challenge. In *Proceedings of the 14th International Symposium on Formal Methods (FM), Lecture Notes in Computer Science*, pages 1–15, Ontario, Canada, August 2006. Springer. 1

[22] Pao-Ann Hsiung and Shang-Wei Lin. Formal Design and Verification of Real-Time Embedded Software. In *Proceedings of the Second Asian Symposium on Programming Languages and Systems*, pages 382–397, Taipei, Taiwan, November 2004. Springer. 4, 22, 24, 25, 138

[23] Jerome Hugues, Bechir Zalila, Laurent Pautet, and Fabrice Kordon. From the prototype to the final embedded system using the ocarina aadl tool suite. *ACM Trans. Embed. Comput. Syst.*, 7(4), 2008. 33, 37

[24] Li Jie, Guo Ruifeng, and Shao Zhixiang. The research of scheduling algorithms in real-time system. In *Proceedings of the 2010 International Conference On Computer and Communication Technologies in Agriculture Engineering*, pages 333–336, Shenyang, China, 2010. IEEE. 17

[25] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. ATL : a QVT-Like Transformation Language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720. ACM, 2006. 84

[26] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Claudio Magalh a, and Roy H. Campbell. Monitoring, Security, and Dynamic Configuration with the DynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed systems platforms*, Middleware 2000, pages 121–143. Springer, 2000. 4, 25, 139

[27] Fatma Krichen. Position paper : Advances in Reconfigurable Distributed Real Time Embedded Systems. In *International workshop on Distributed Architecture modeling for Novel component based Embedded systems*, pages 273–278, Tozeur, Tunisia, May 2010. IEEE. 9

[28] Fatma Krichen, Amal Gassara, Bechir Zalila, Brahim Hamid, and Mohamed Jmaiel. Modélisation et Vérification des Systèmes Embarqués Temps Réel Reconfigurables. In *6ème Conférence Internationale Francophone sur les Architectures Logicielles (CAL)*, 2012. 58

[29] Fatma Krichen, Amal Gassara, Bechir Zalila, and Mohamed Jmaiel. Towards a Verification Approach for Reconfigurable Embedded Systems. In *Proceedings of the 17th IEEE Symposium on Computers and Communications*, pages 750–752, Cappadocia, Turkey, july 2012. IEEE. 57

[30] Fatma Krichen, Amal Ghorbel, Brahim Hamid, and Bechir Zalila. An MDE-Based Approach for Reconfigurable Embedded Systems. In *Proceedings of the 21st IEEE International Conference on Collaboration Technologies and Infrastructures*, pages 78–83, Toulouse, France, juin 2012. IEEE Computer Society. 5, 40

[31] Fatma Krichen, Brahim Hamid, Bechir Zalila, and Bernard Coulette. Designing Dynamic Reconfiguration of Distributed Real Time Embedded Systems.

In *Proceedings of the 10th annual international conference on New Technologies of Distributed Systems*, pages 249–254, Tozeur, Tunisie, June 2010. IEEE Computer Society. 51

[32] Fatma Krichen, Brahim Hamid, Bechir Zalila, and Mohamed Jmaiel. Towards a Model-Based Approach for Reconfigurable DRE Systems. In *Proceedings of the 5th European Conference on Software Architecture*, pages 295–302, Essen, Germany, September 2011. Springer. 6, 45, 51

[33] Fatma Krichen, Brahim Hamid, Bechir Zalila, and Mohamed Jmaiel. Design-Time Verification of Reconfigurable Real-Time Embedded Systems. In *Proceedings of the 9th IEEE International Conference on Embedded Software and Systems*, pages 1487–1494, Liverpool, UK, june 2012. IEEE. 57

[34] Fatma Krichen, Brahim Hamid, Bechir Zalila, Mohamed Jmaiel, and Bernard Coulette. Development of Reconfigurable Distributed Embedded Systems with a Model-Driven Approach. *Journal of Concurrency and Computation :Practice and Experience*, 2013. To appear. 40

[35] Fatma Krichen, Bechir Zalila, Mohamed Jmaiel, and Brahim Hamid. A Middleware for Reconfigurable Distributed Real-Time Embedded Systems. In *Proceedings of the ACIS International Conference on Software Engineering Research, Management and Applications SERA (selected papers)*, Studies in Computational Intelligence, pages 81–96, Shanghai, China, 2012. Springer. 72

[36] Jagun Kwon, Andy Wellings, and Steve King. Ravenscar-Java : a High Integrity Profile for Real-Time Java. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 131–140, New York, NY, USA, 2002. ACM. 28

[37] Gilles Lasnier, Bechir Zalila, Laurent Pautet, and Jérôme Hugues. Ocarina : An environment for aadl models analysis and automatic code generation for high integrity applications. In *Proceedings of the 14th Ada-Europe International Conference on Reliable Software Technologies*, pages 237–250, Brest, France, 2009. Springer. 33, 37, 138

[38] Joseph Y.-T. Leung and M. L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Inf. Process. Lett.*, 11(3) :115–118, 1980. 17

[39] Chang L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 20(1) :46–61, 1973. 16, 17, 59

[40] Gabor Madl, Sherif Abdelwahed, and Douglas C. Schmidt. Verifying distributed real-time properties of embedded systems via graph transformations and model checking. *Real-Time Systems*, 33(1–3) :77–100, 2006. 21, 24, 25

[41] OMG. Real-time CORBA Specification. http://www.omg.org, January 2005. 21, 26

[42] OMG. UML Profile for Schedulability, Performance, and Time Specification. http://www.omg.org/technology/documents/formal/schedulability.htm, January 2005. 11

[43] OMG. Deployment and Configuration of Component-based Distributed Applications Specification, v4.0. http://www.omg.org/cgi-bin/doc?formal/06-04-02, April 2006. 32

[44] OMG. CORBA Specification, Version 3.1. Part 3 : CORBA Component Model. http://www.omg.org/technology/documents/corba_spec_catalog.htm#CCM, October 2008. 26, 32, 33

[45] OMG. A UML Profile for MARTE : Modeling and Analysis of Real-Time Embedded systems. www.omgmarte.org/Documents/Specifications/08-06-09.pdf, 2009. xiii, 3, 6, 11, 12, 14, 15, 31, 42, 44, 46, 138

[46] OMG. OMG Unified Modeling Language (OMG UML), Superstructure. http://www.omg.org/spec/UML/2.2/Superstructure, February 2009. 35

[47] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. http ://www.omg.org/spec/QVT/1.1/PDF/, 2011. 31

[48] Martin Ouimet, Guillaume Berteau, and Kristina Lundqvist. Modeling an Electronic Throttle Controller Using the Timed Abstract State Machine Language and Toolset. In *Workshops and Symposia at MoDELS in Software Engineering*, pages 32–41, Genoa, Italy, October 2006. Springer. 20, 24

[49] Martin Ouimet and Kristina Lundqvist. The TASM Toolset : Specification, Simulation, and Formal Verification of Real-Time Systems (Tool Paper). In

Proceedings of the 19th International Conference on Computer Aided Verification, pages 126–130, Berlin, Germany, July 2007. Springer. 4, 20, 24, 25, 138

[50] Juraj Polakovic, Sebastien Mazare, Jean-Bernard Stefani, and Pierre-Charles David. Experience with Safe Dynamic Reconfigurations in Component-Based Embedded Systems. In *Proceedings of the International Symposium in Component-Based Software Engineering*, pages 242–257. Springer, 2007. 1, 14, 15

[51] Juraj Polakovic and Jean-Bernard Stefani. Architecting reconfigurable component-based operating systems. *Journal of Systems Architecture - Embedded Systems Design*, 54(6) :562–575, 2008. 13

[52] SAE. Architecture Analysis & Design Language (AADL). `http://www.sae.org/technical/standards/AS5506A`, January 2009. 3, 6, 10, 14, 15, 33, 111, 138

[53] Douglas C. Schmidt. Guest Editor's Introduction : Model-Driven Engineering. *Computer*, 39(2) :25–31, 2006. 3

[54] Douglas C. Schmidt and Chris Cleeland. *Applying a pattern language to develop extensible ORB middleware*. Cambridge University Press, 2001. 25, 28

[55] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols : An Approach to real-time Synchronization. *IEEE transactions on computers*, 39(9) :1175–1185, September 1990. 64, 93

[56] Frank Singhoff, Jérôme Legrand, L. Nana, and Lionel Marcé. Cheddar : a Flexible Real Time Scheduling Framework. In *Proceedings of the 2004 Annual ACM SIGAda International Conference on Ada : The Engineering of Correct and Reliable Software for Real-Time & Distributed Systems Using Ada and Related Technologies*, pages 1–8, New York, NY, USA, 2004. ACM. 19, 24, 25, 33, 59, 139

[57] D. Steinberg, F. Budinsky, M. Paternostro, and Ed Merks. *EMF : Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009. 84

[58] Venkita Subramonian, Gan Deng, Christopher Gill, Jaiganesh Balasubramanian, Liang-Jui Shen, William Otte, Douglas C. Schmidt, Aniruddha Gokhale, and Nanbor Wang. The Design and Performance of Component Middleware

for QoS-enabled Deployment and Configuration of DRE Systems. *Journal of Systems and Software*, 80 :668–677, May 2007. 4, 26, 139

[59] Thomas Vergnaud, Jérôme Hugues, Laurent Pautet, and Fabrice Kordon. PolyORB : A Schizophrenic Middleware to Build Versatile Reliable Distributed Applications. In *Proceedings of the 9th Ada-Europe International Conference on Reliable Software Technologies*, pages 106–119. Springer, June 2004. 28

[60] Farn Wang. Formal Verification of Timed Systems : A Survey and Perspective. In *Proceedings of the IEEE*, pages 1283–1305. IEEE, 2004. 4

[61] B. Zalila. *Configuration et déploiement d'applications temps-réel réparties embarquées à l'aide d'un langage de description d'architecture*. PhD thesis, École Nationale Supérieure des Télécommunications, 2008. xix, 34, 71

[62] Bechir Zalila. *Configuration et déploiement d'applications temps-réel réparties embarqués à l'aide d'un langage de description d'architecture*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, 2008. xiii, xix, 34, 46, 52

[63] Bechir Zalila, Laurent Pautet, and Jérôme Hugues. Towards Automatic Middleware Generation. In *Proceedings of the International Symposium on Object-oriented Real-time distributed Computing*, pages 221–228. IEEE, 2008. 28, 69

Fatma Krichen
Directeurs de thèse : Bernard Coulette & Brahim Hamid – Université de Toulouse
Mohamed Jmaiel & Bechir Zalila – Université de sfax
Lieu et date de soutenance : Toulouse, le 16 Septembre 2013

Titre : Architectures Logicielles à Composants Reconfigurables pour les Systèmes TR²E
Résumé :
Un système logiciel embarqué est dit reconfigurable, s'il peut modifier son comportement ou son architecture selon l'évolution des exigences de son contexte d'utilisation et la variation des contraintes de son environnement d'exécution. La croissance constante de la complexité afférente et l'autonomie indispensable à la gestion des systèmes logiciels embarqués rendent la reconfiguration de plus en plus importante. Les défis concernent autant le niveau modèle de conception que le niveau environnement et support d'exécution. Les contributions de ce travail portent sur la reconfiguration dynamique guidée par les modèles dans le processus de développement des systèmes logiciels embarqués. Elles ciblent à la fois le niveau modélisation et le niveau plate-forme d'exécution. Par ailleurs, nous proposons une approche basée sur l'ingénierie dirigée par les modèles permettant le passage automatisé et fiable des modèles vers l'implantation, sans rupture de la chaîne de production.
Mots-clés : reconfiguration dynamique, systèmes TR²E, propriétés non-fonctionnelles, IDM, intergiciel.

Title : Reconfigurable components software architectures of distributed real-time embedded systems
Abstract :
An embedded software system is reconfigurable when it can modify its behavior or its architecture. The reconfigurations are launched according to the evolution of context requirements and the variation of execution environment constraints. The constant growth of the complexity in embedded systems makes the reconfiguration more important and more difficult to achieve. The challenges concern as much the design model level as the runtime support level. The development of these systems according to the traditional processes is not more applicable in this context. New methods are necessary to conceive and to supply reconfigurable embedded software architectures. We propose a model driven approach that enables to specify dynamic embedded software architectures with respect to non-functional properties. We also propose a runtime support that enables to perform dynamic embedded applications generated from a high level description.
Keywords : dynamic reconfiguration, distributed real time embedded systems, non-functional properties, MDE, middleware.

Ecole doctorale : Mathématiques Informatique Télécommunications (MITT)
Unité de recherche : Institut de Recherche en Informatique de Toulouse (IRIT)

IRIT (Institut de Recherche en Informatique de Toulouse), Université Paul Sabatier, 118 Route de Narbonne, F-31062 TOULOUSE CEDEX 9